SESSION 7

iOS DEVELOPMENT

ADVANCED IOS

- It’s time to start thinking about courses for next quarter!
- Advanced iOS (MPCS 51032) will be offered on Tuesday evenings in the Spring

ADVANCED IOS

- Topics that have been covered in the past
  - Integrating with Siri
  - Building apps for Apple Watch
  - Building games with SpriteKit
  - Building AR experiences

CASE STUDIES

CASE STUDY PRESENTATIONS

- Case studies will be on Wednesday, March 4
- 5 minute presentation on an app in the App Store
  - Plus a few extra minutes for Q&A
- You may work individually or with one partner

CASE STUDY PRESENTATIONS

- Add your name and the app you're presenting on to the sign-up sheet
- Don't sign up for an app someone else is presenting on!

CASE STUDY PRESENTATIONS

- Basic information:
    - What is the basic functionality and purpose of the app?
    - Who is the developer? What role does the app play in their business model?
    - How long has the app been around? How does it rank on the App Store?

CASE STUDY PRESENTATIONS

- Competition:
    - Who are the app's primary competitors?
    - How does it set itself apart from the competition?

CASE STUDY PRESENTATIONS

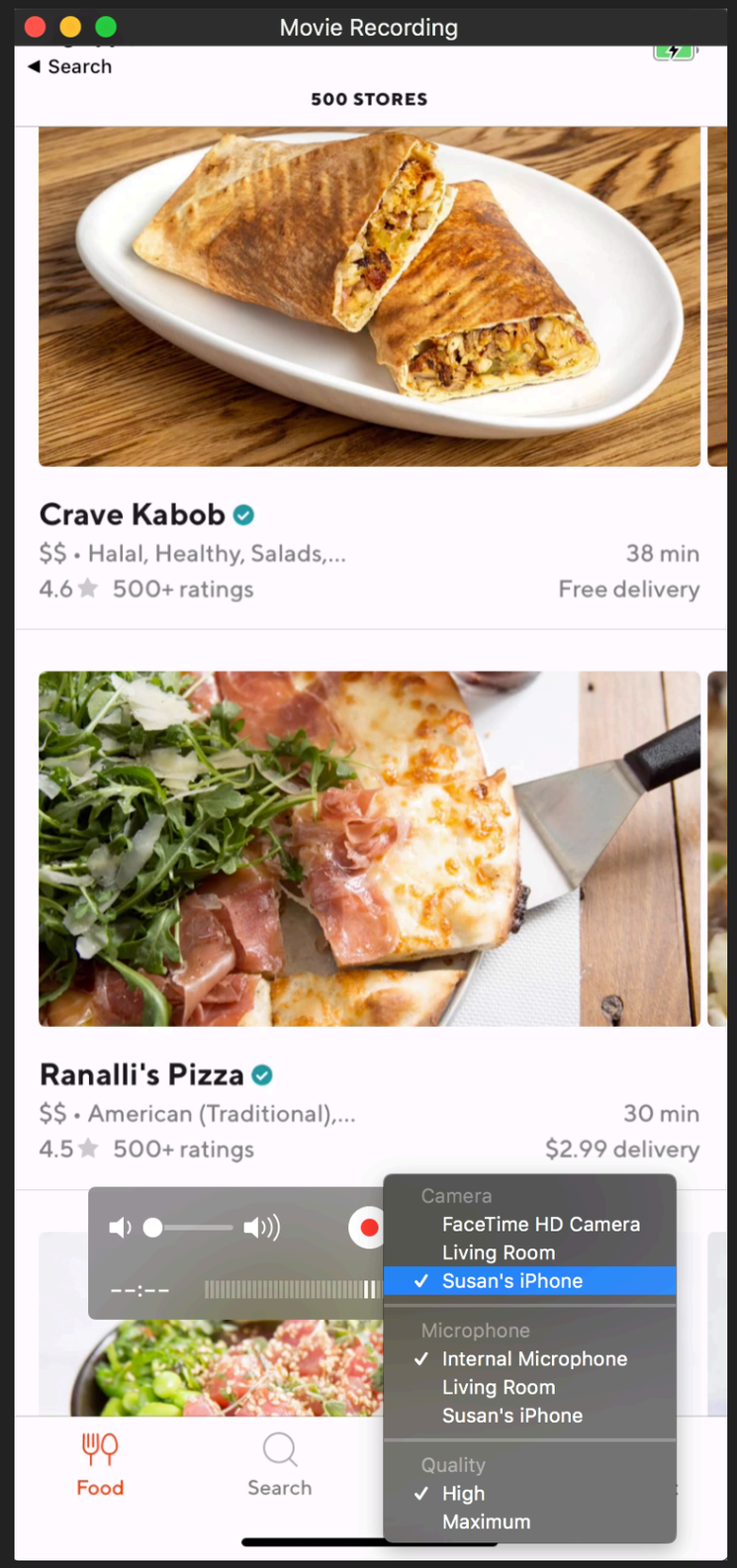
- Functionality:
    - Describe the app's central functionality
    - Show screenshots and / or videos
    - What do you like about this app? What could be improved?

CASE STUDY PRESENTATIONS

- Technical Implementation:
    - Tell us which UI components and frameworks you think were used
    - Choose a feature that looks challenging to implement and describe how it might have been done

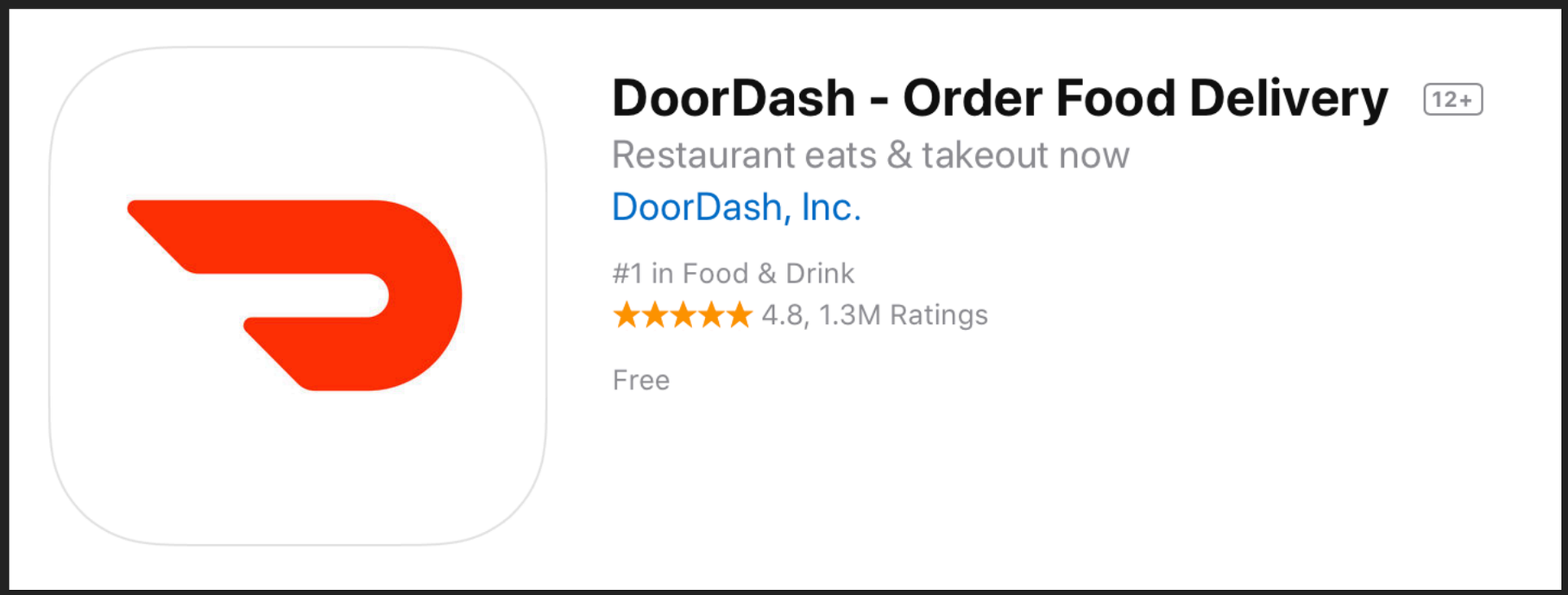
CREATING VIDEOS

- To record a video of an app:
  - Connect your iOS device to your laptop
  - Open Quicktime and select “New Movie Recording”
  - Choose your device from the dropdown next to the record button



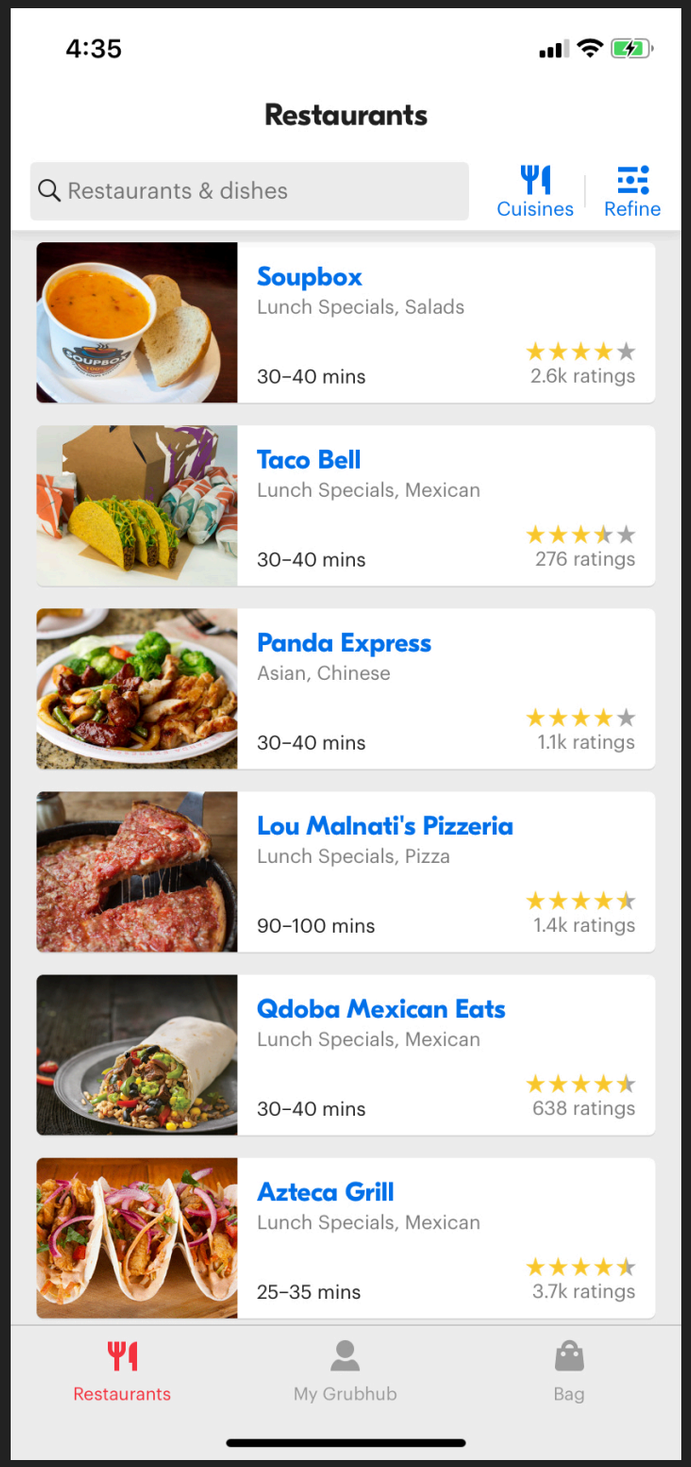
DOORDASH

- Food delivery service
- Founded in 2013 by four students at Stanford University
- Ranked #1 in Food & Drink category on the App Store

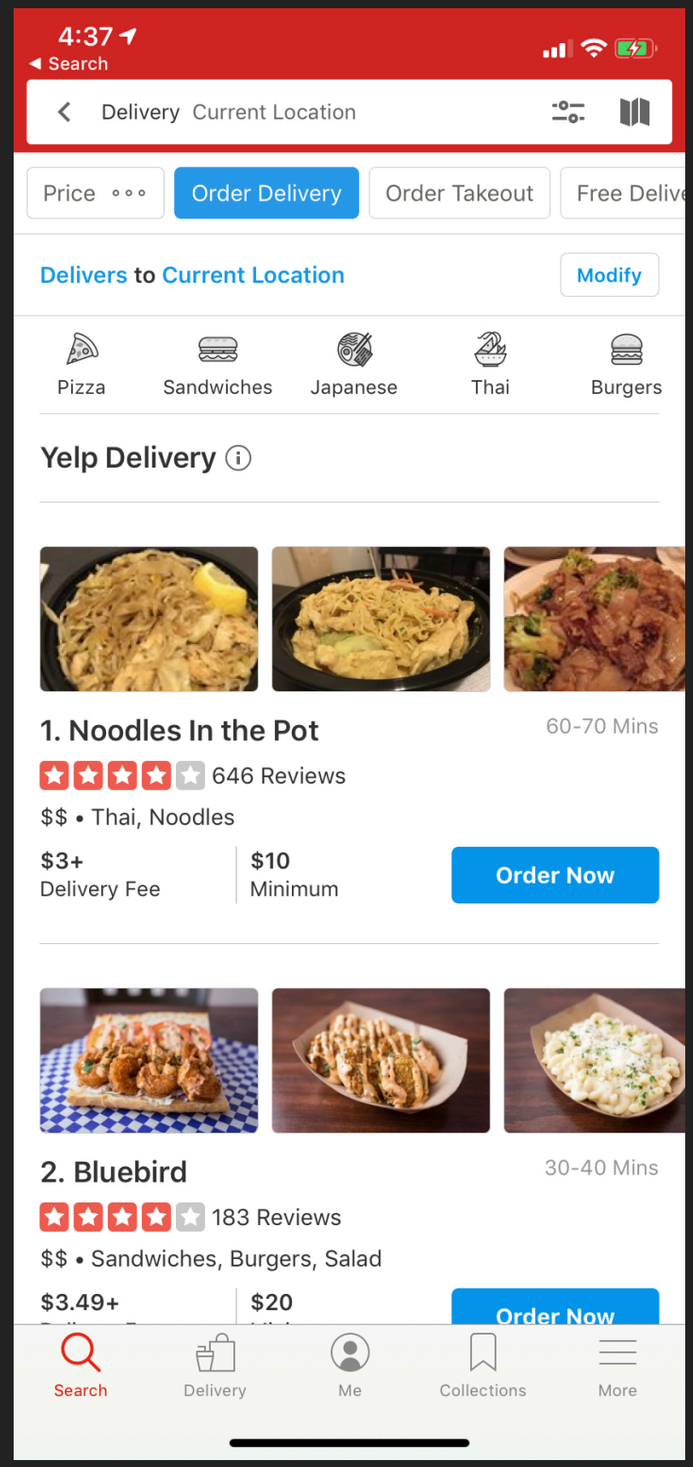


DOORDASH

- Competition:
  - Grubhub
  - Yelp
  - Postmates
  - Uber Eats



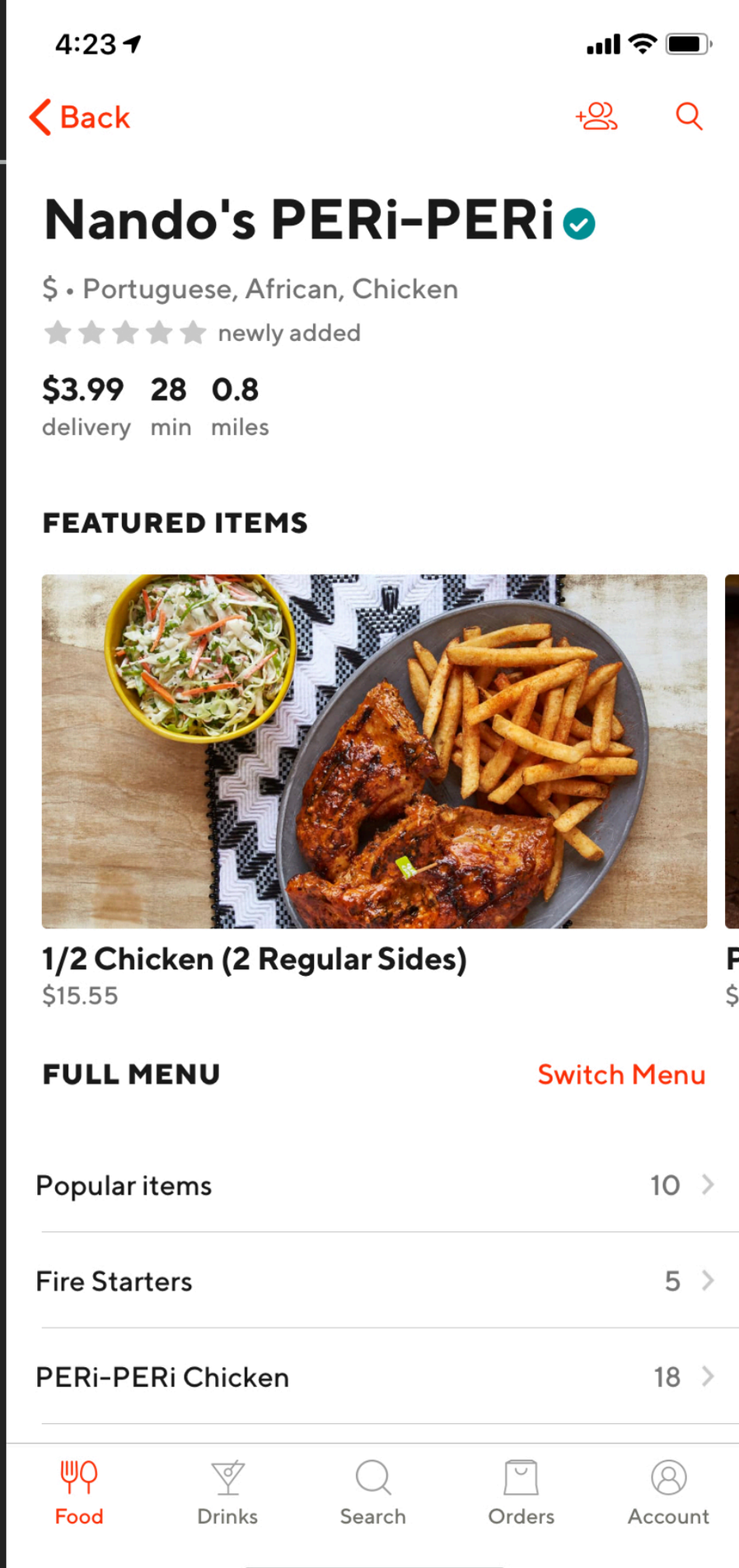
Grubhub



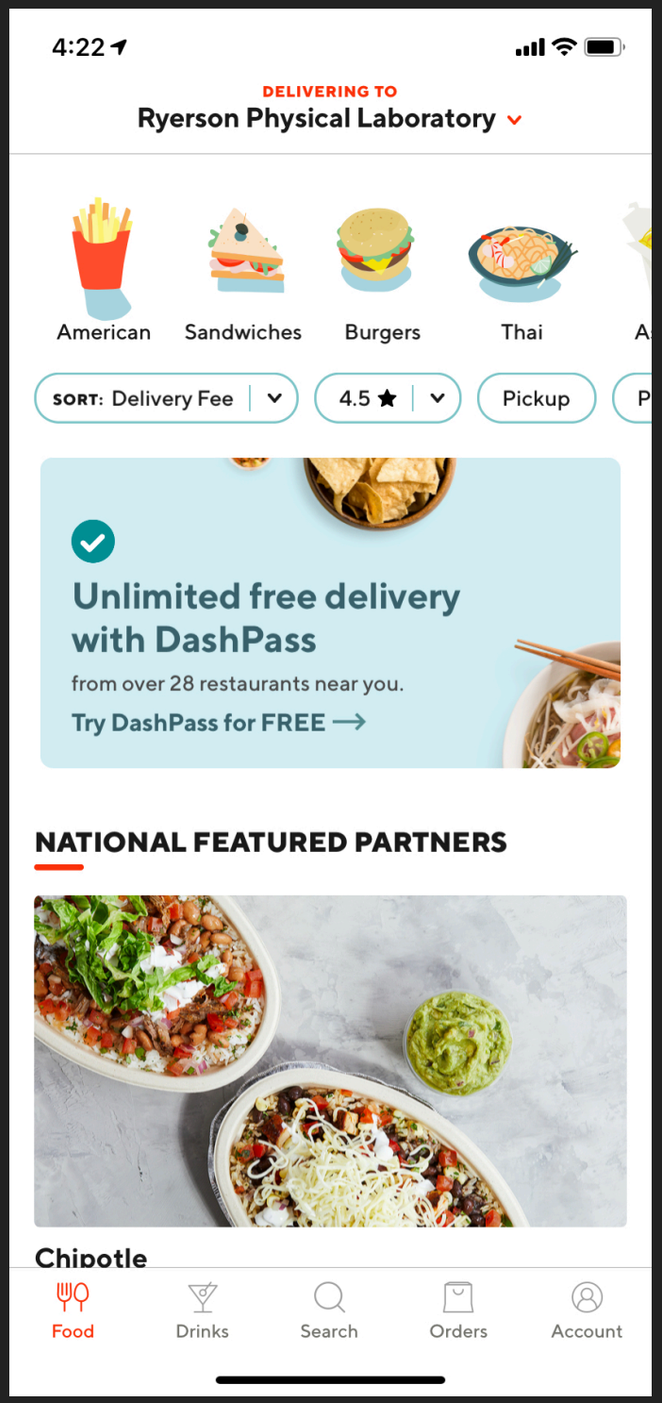
Yelp

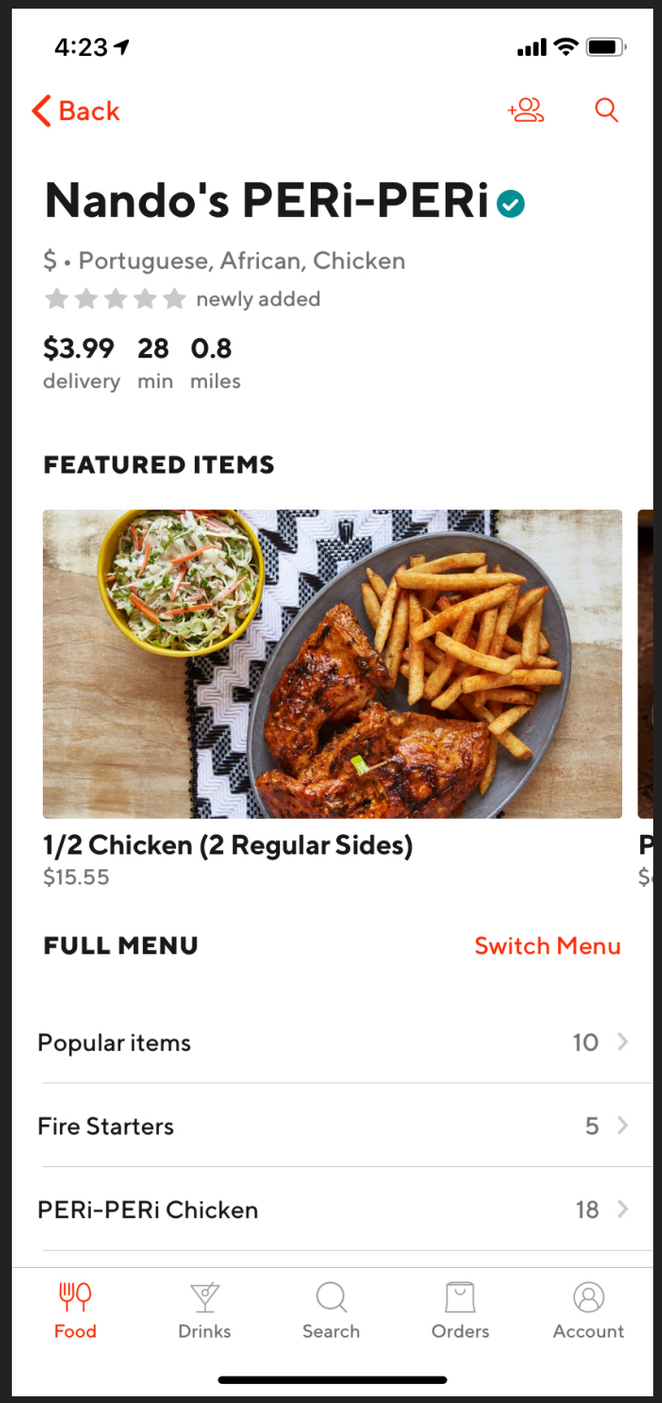
DOORDASH

- Uses Apple's preferred design patterns, like large titles
- More intuitive and visually pleasing than it's competitors



DOORDASH

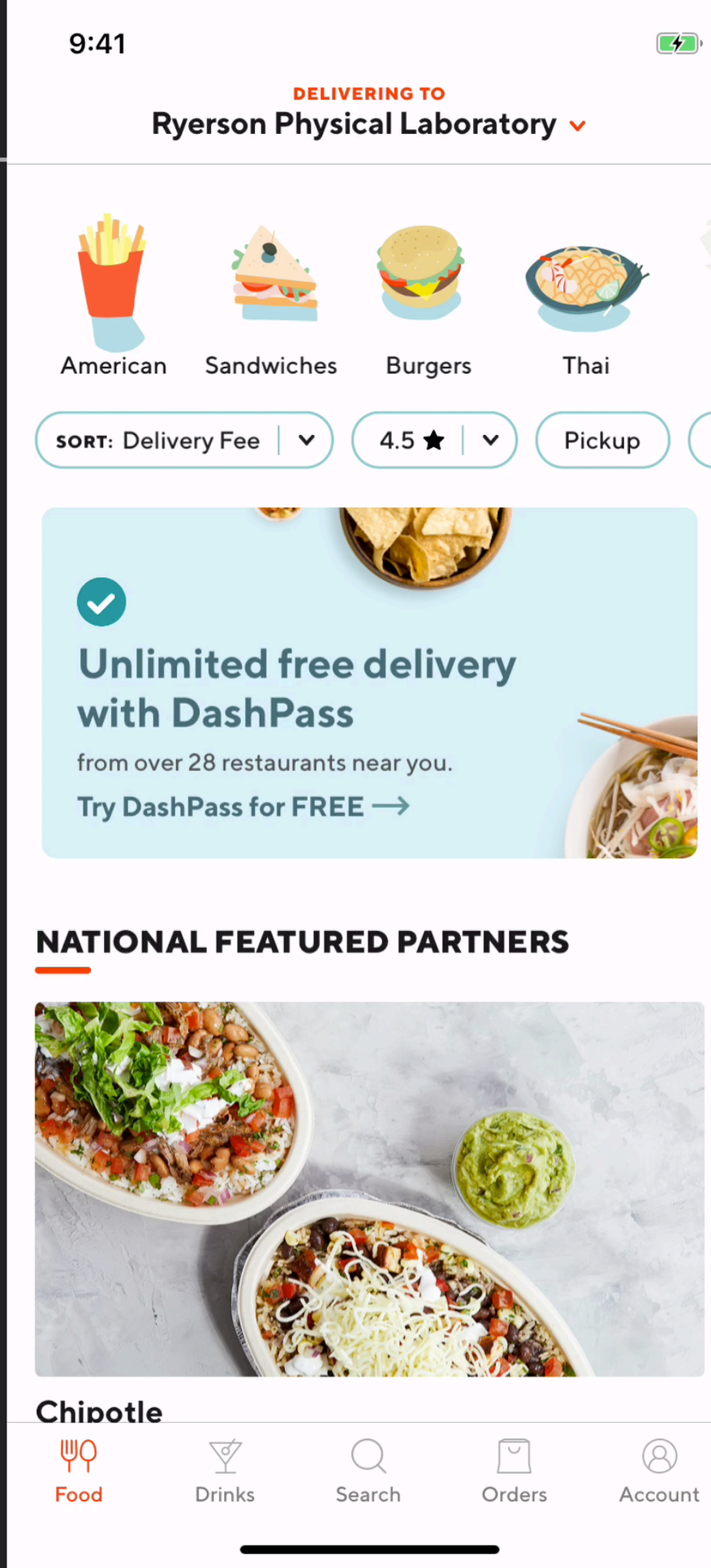

4:22
DELIVERING TO
Ryerson Physical Laboratory
American
Sandwiches
Burgers
Thai
SORT: Delivery Fee
4.5
Pickup
Unlimited free delivery with DashPass
from over 28 restaurants near you.
Try DashPass for FREE
NATIONAL FEATURED PARTNERS
Chipotle
Food
Drinks
Search
Orders
Account


4:23
Back
Nando's PERi-PERi
$ • Portuguese, African, Chicken
newly added
$3.99 delivery
28 min
0.8 miles
FEATURED ITEMS
1/2 Chicken (2 Regular Sides)
$15.55
FULL MENU
Switch Menu
Popular items 10
Fire Starters 5
PERi-PERi Chicken 18
Food
Drinks
Search
Orders
Account


4:22
Search
Search DoorDash
RECENTS
Native Foods
Vegetarian
Tandoor Char House
CUISINES
African
American
Asian
Barbecue
Burgers
Cafes
Caribbean
Catering
Chinese
Food
Drinks
Search
Orders
Account

DOORDASH

- Full-screen view controller presented modally
- Dark gray view with alpha < 1.0
- Animates on-screen in viewDidAppear



AN INTRODUCTION TO

SWIFTUI

INTRODUCING SWIFTUI

- Apple introduced SwiftUI at WWDC 2019
  - Swift-only UI framework
  - Alternative to UIKit

COMPARISON

- UIKit
  - Obj C and Swift
  - Storyboards
  - Any version of iOS
  - Imperative
- SwiftUI
  - Swift only
  - Live Previews
  - iOS 13 and above
  - Declarative

DECLARATIVE VS. IMPERATIVE

- With imperative programming, developers give step-by-step instructions
    - “First make a label. Then add it as a subview…”
- With declarative programming, developers describe the desired end-state
    - “I want a label in the center of the screen.”

IMPERATIVE EXAMPLE

- Imagine teaching someone who doesn’t know how to cook to make pasta over the phone:
    - 1. Boil water on the stove.
    - 2. Add the pasta to the water.
    - 3. Set a timer for 10 minutes.
    - 4. Check if the pasta is done.
    - 5. Etc…

DECLARATIVE EXAMPLE

- Imagine ordering pasta at a restaurant:
    - “I’d like spaghetti with meatballs and extra Parmesan cheese.”

VIEWS

- Views are the building blocks of user interfaces
  - In UIKit, we use `UIView`
  - In SwiftUI, we use `View`

VIEWS

- Here is the code for a basic view in SwiftUI

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Hello, World!")
    }
}
```

VIEWS

- Here is the code for a basic view in SwiftUI

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Hello, World!")
    }
}
```

VIEWS

- Here is the code for a basic view in SwiftUI

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Hello, World!")
    }
}
```

View is a protocol, not a class

VIEWS

- Here is the code for a basic view in SwiftUI

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Hello, World!")
    }
}
```

Every view has a **body**, which is itself a view

VIEWS

- Here is the code for a basic view in SwiftUI

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Hello, World!")
    }
}
```

Notice that we don't need to call **addSubview**

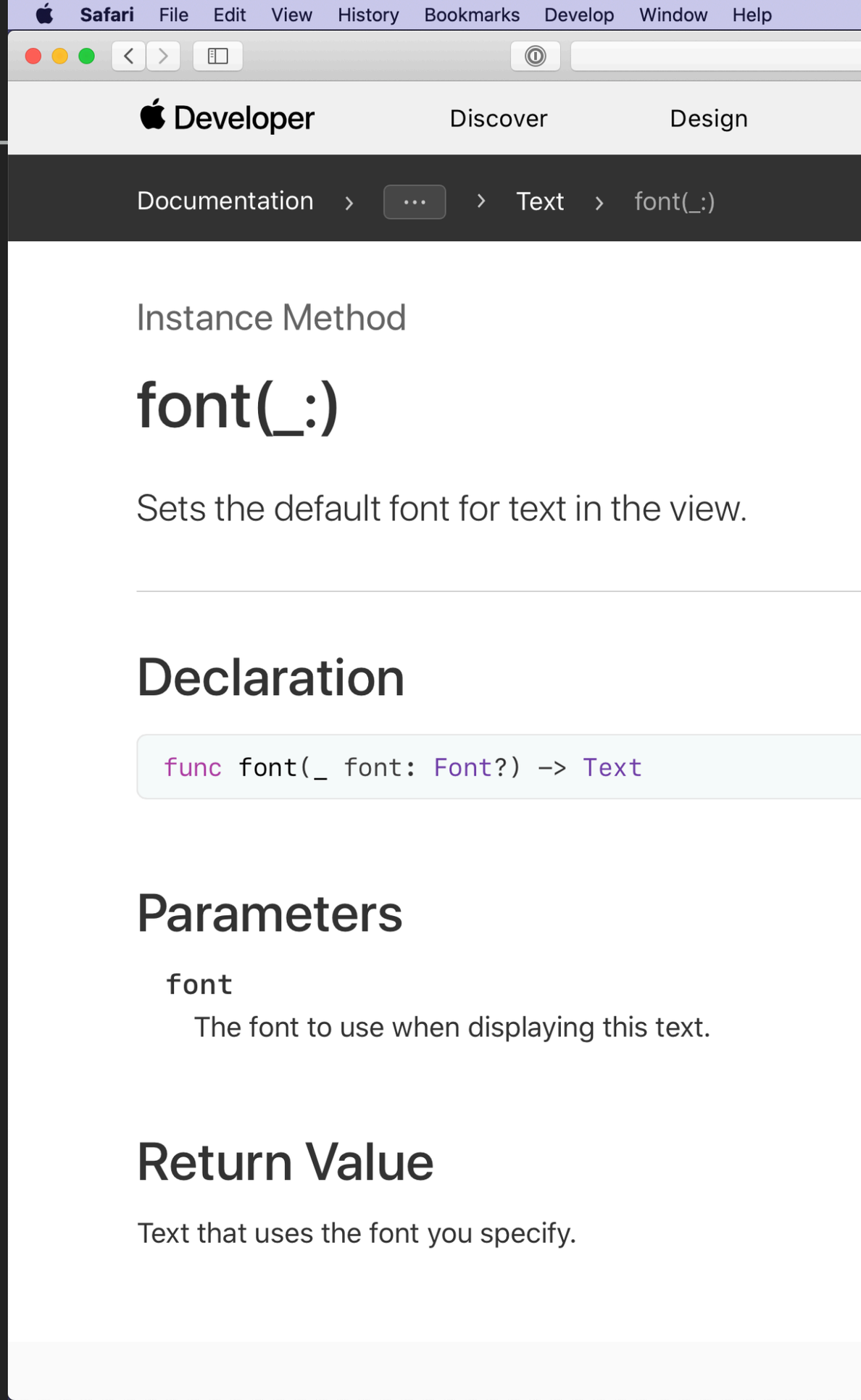
VIEW MODIFIERS

- Change how a view renders using view modifiers

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Hello, World!")
            .font(Font.largeTitle)
    }
}
```

VIEW MODIFIERS

- The `font` modifier is a method on `Text`
- It takes font parameter and returns an instance of Text



VIEW MODIFIERS

- You can chain modifiers together, like this:

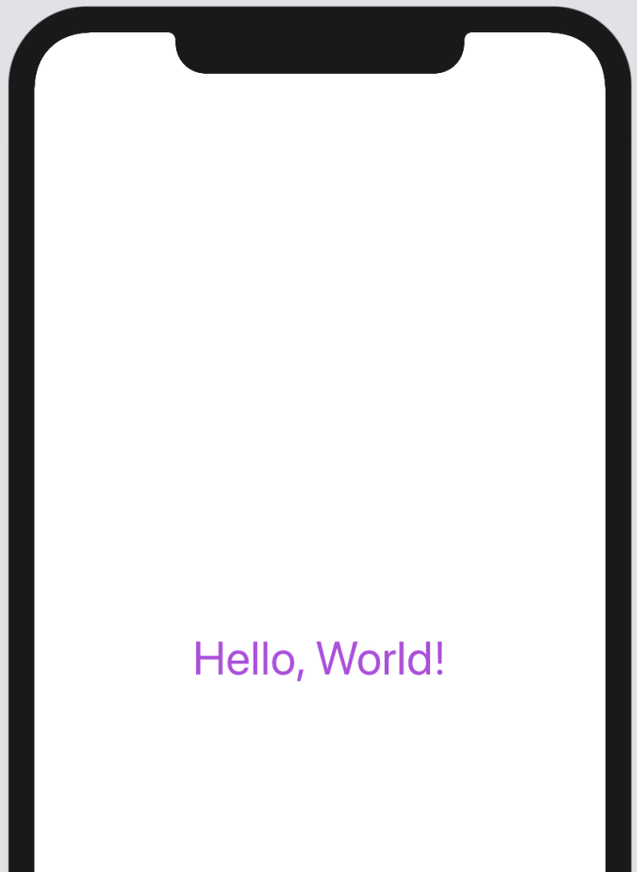
```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Hello, World!")
            .font(Font.largeTitle)
            .foregroundColor(Color.purple)
    }
}
```

Chaining view modifiers

VIEW MODIFIERS

- The `PreviewProvider` protocol allows you to preview views as you design them


```swift
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Hello, World!")
            .font(.largeTitle)
            .foregroundColor(Color.purple)
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```



VIEW MODIFIERS

- The preview updates automatically as you change your code

Xcode File Edit View Find Navigate Editor Product Debug Source Control Window Help

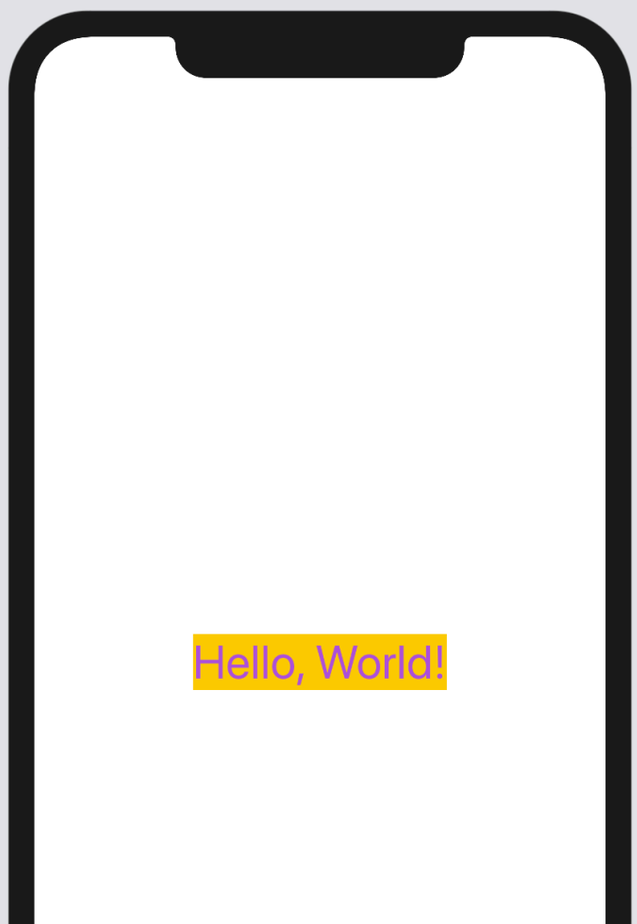
8% Feb 17 5:06 PM

SwiftUIDemo 〉 iPhone 11 Pro Max

SwiftUIDemo | Preview SwiftUIDemo: **Succeeded** | Today at 1:25 PM

SwiftUIDemo 〉 SwiftUIDemo 〉 ContentView.swift 〉 No Selection

```
8
9   import SwiftUI
10
11  struct ContentView: View {
12      var body: some View {
13          Text("Hello, World!")
14              .font(.largeTitle)
15              .foregroundColor(Color.purple)
16              .background(Color.yellow)
17      }
18  }
19
20  struct ContentView_Previews: PreviewProvider {
21      static var previews: some View {
22          ContentView()
23      }
24  }
25
```

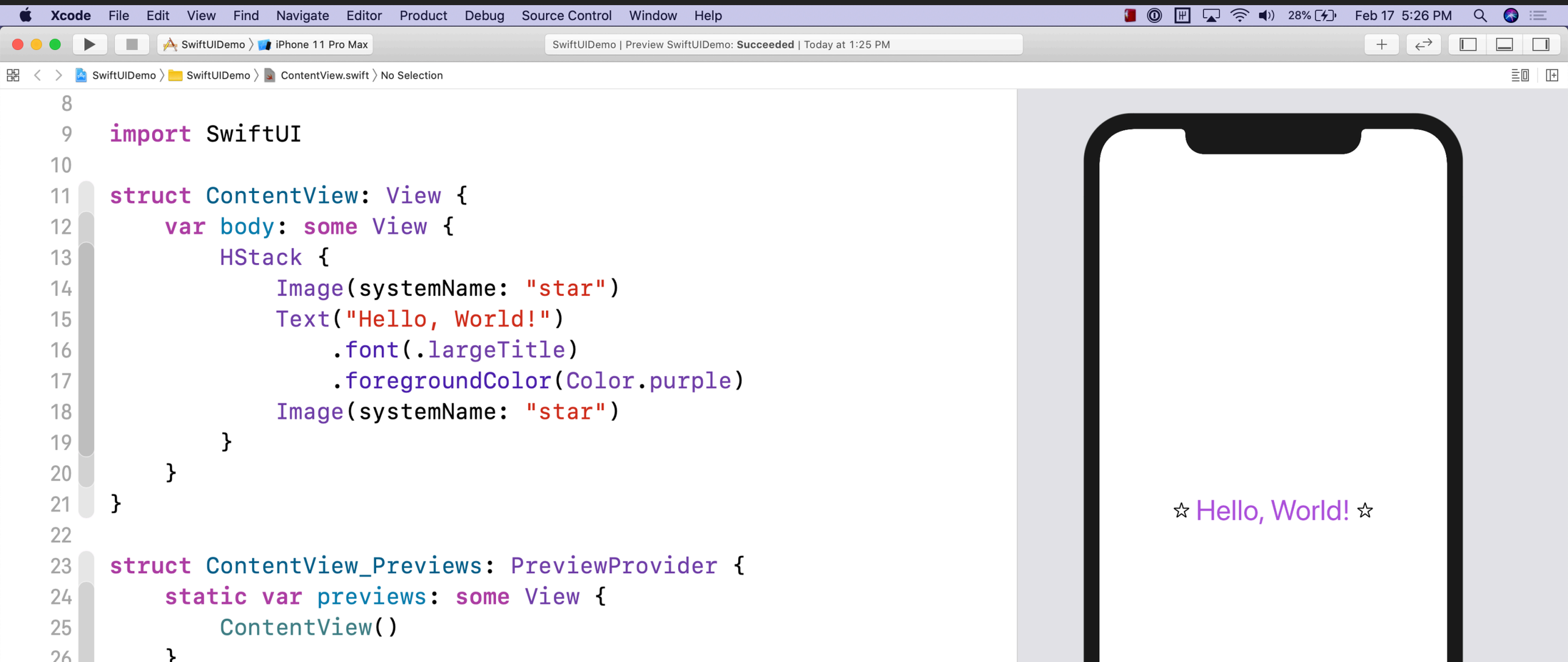


VIEW MODIFIERS

- SwiftUI provides three container views that arrange other views into stacks
    - HStack (horizontal)
    - VStack (vertical)
    - ZStack (overlay)

VIEW MODIFIERS

- HStack arranges its child views horizontally



VIEW MODIFIERS

- VStack arranges its child views vertically

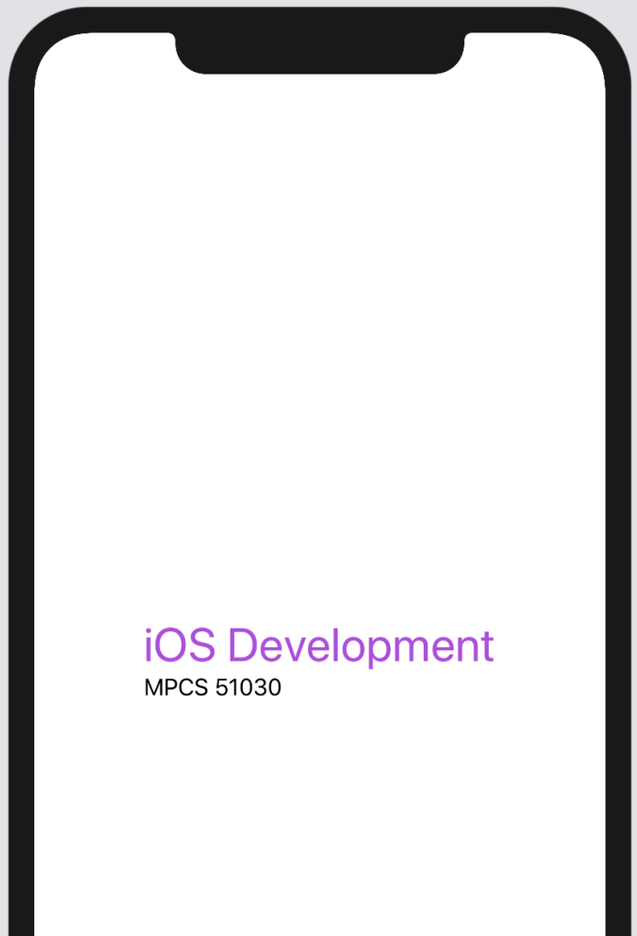
Xcode File Edit View Find Navigate Editor Product Debug Source Control Window Help 29% Feb 17 5:27 PM

SwiftUIDemo 〉 iPhone 11 Pro Max | SwiftUIDemo | Preview SwiftUIDemo: **Succeeded** | Today at 1:25 PM

SwiftUIDemo 〉 SwiftUIDemo 〉 ContentView.swift 〉 No Selection

```
8
9   import SwiftUI
10
11  struct ContentView: View {
12      var body: some View {
13          VStack(alignment: .leading) {
14              Text("iOS Development")
15                  .font(.largeTitle)
16                  .foregroundColor(Color.purple)
17              Text("MPCS 51030")
18          }
19      }
20  }
21
22  struct ContentView_Previews: PreviewProvider {
23      static var previews: some View {
24          ContentView()
25      }
26  }
```

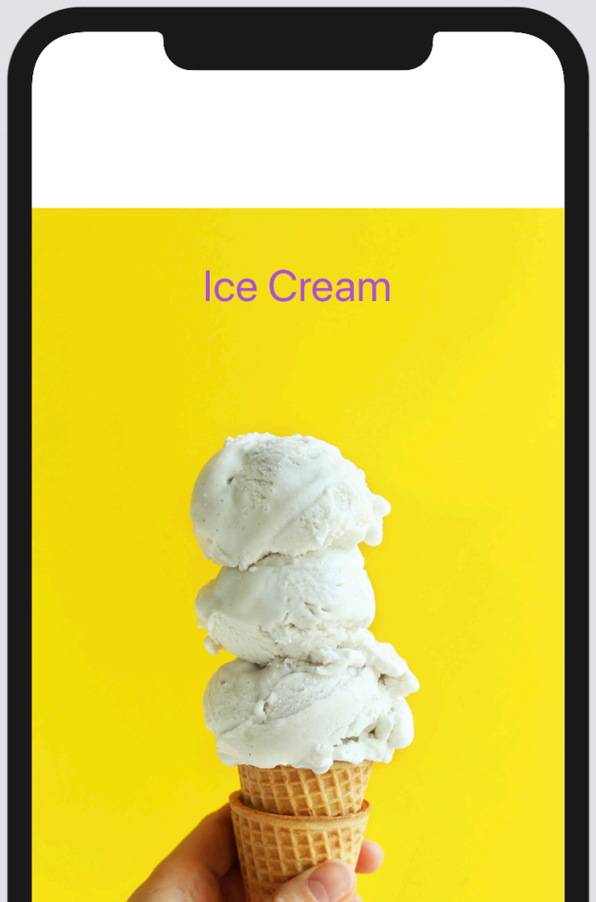
Leading alignment



VIEW MODIFIERS

- ZStack overlays its child views on top of each other

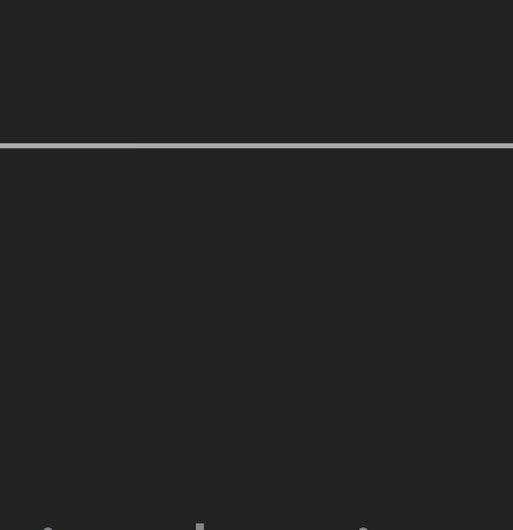

```swift
import SwiftUI

struct ContentView: View {
    var body: some View {
        ZStack(alignment: .top) {
            Image("ice-cream")
                .resizable()
                .aspectRatio(contentMode: .fit)
            Text("Ice Cream")
                .font(.largeTitle)
                .foregroundColor(Color.purple)
                .padding(.top, 40)
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
```

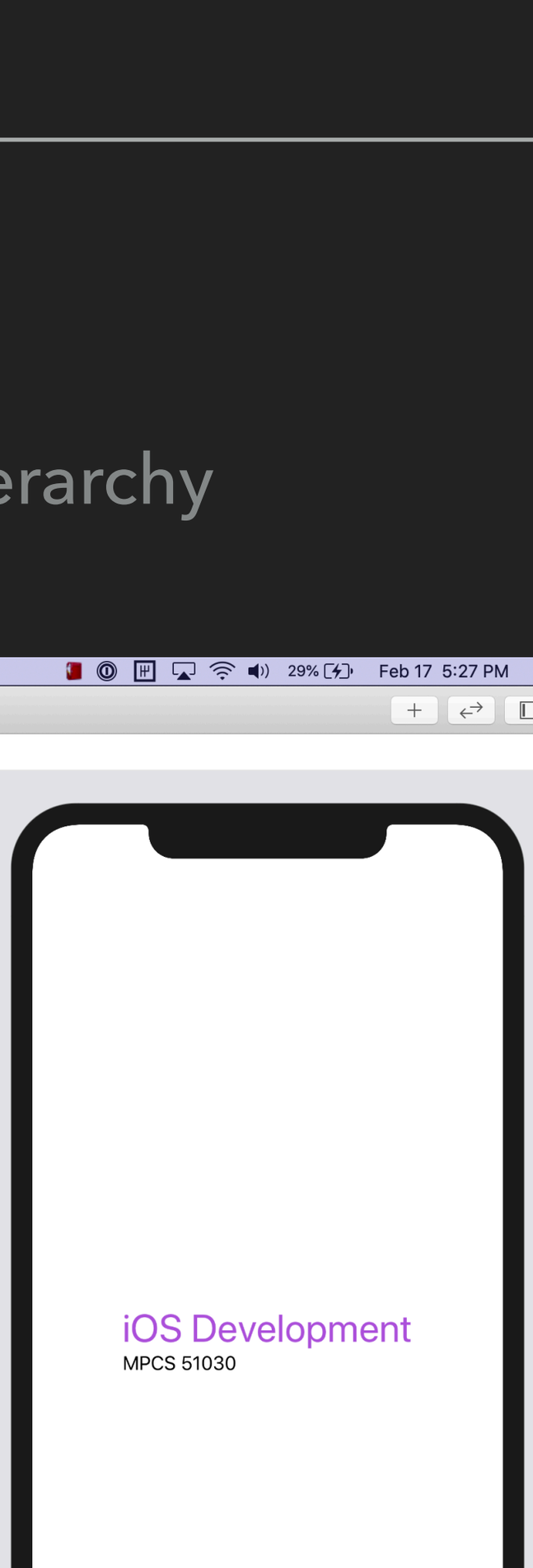


VIEW MODIFIERS

- Notice how the code mimics the view hierarchy


```swift
import SwiftUI

struct ContentView: View {
    var body: some View {
        VStack(alignment: .leading) {
            Text("iOS Development")
                .font(.largeTitle)
                .foregroundColor(Color.purple)
            Text("MPCS 51030")
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```





LISTS AND NAVIGATION

SWIFTUI

LISTS

- `List` allows you to present multiple rows of data

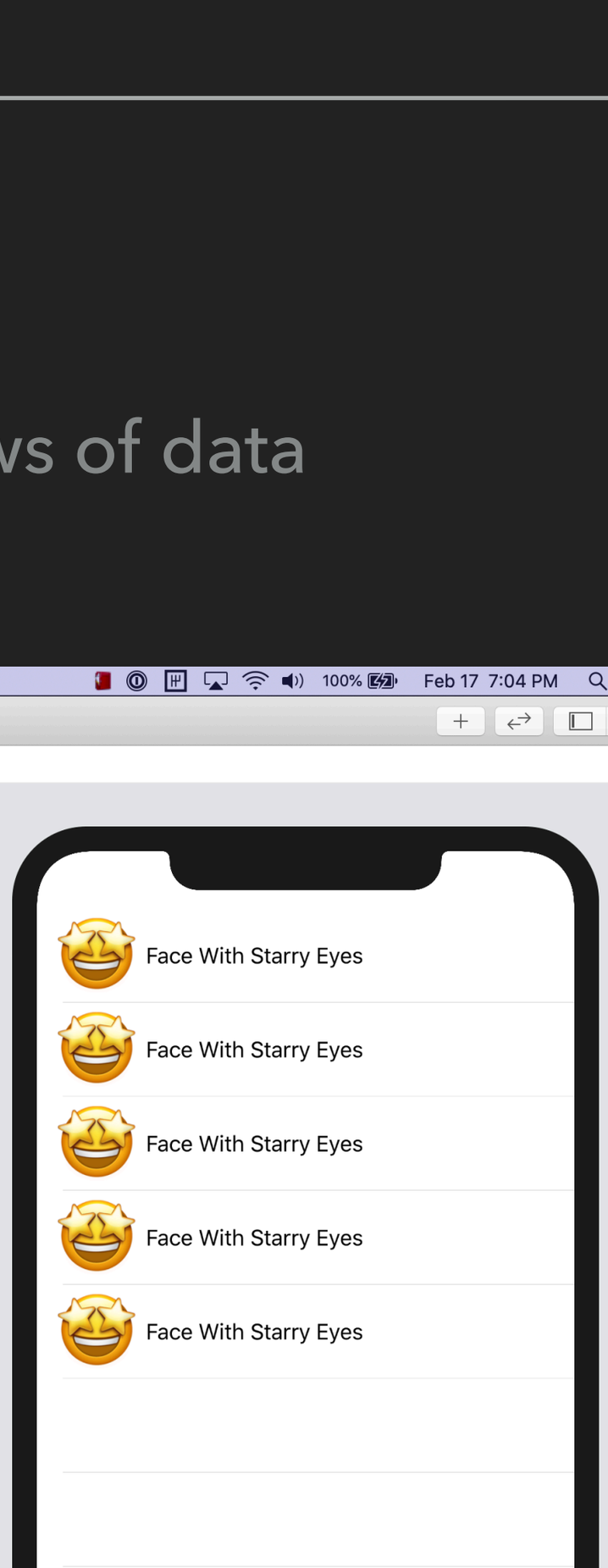
Xcode File Edit View Find Navigate Editor Product Debug Source Control Window Help 100% Feb 17 7:04 PM

SwiftUIDemo 〉 iPhone 11 Pro Max | Running SwiftUIDemo on iPhone 11 Pro Max

SwiftUIDemo 〉 SwiftUIDemo 〉 ContentView.swift 〉 No Selection

```
8
9   import SwiftUI
10
11  struct ContentView: View {
12      var body: some View {
13          List(0..<5) { item in
14              Image("grinning-face-with-star-eyes")
15                  .resizable()
16                  .frame(width: 60, height: 60)
17              Text("Face With Starry Eyes")
18          }
19      }
20  }
21
22  struct ContentView_Previews: PreviewProvider {
23      static var previews: some View {
```

Looks like a table view!

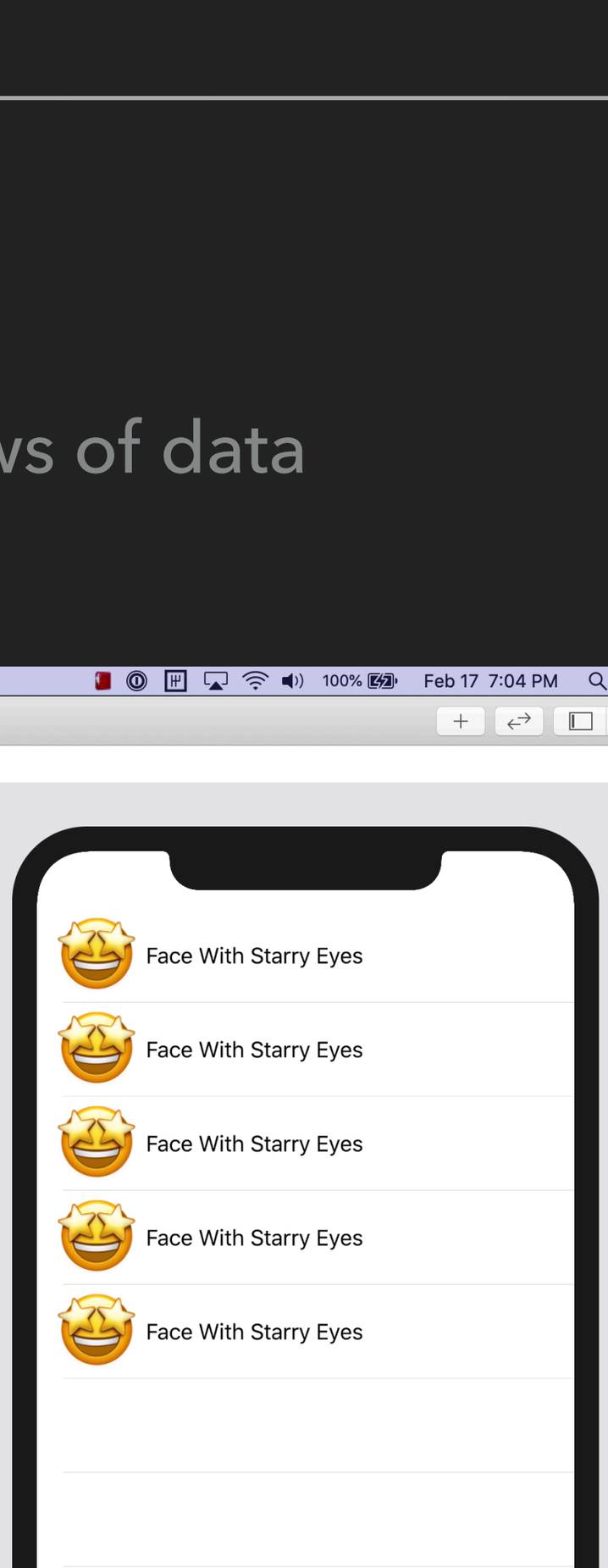


LISTS

- `List` allows you to present multiple rows of data


```
8
9   import SwiftUI
10
11  struct ContentView: View {
12      var body: some View {
13          List(0..<5) { item in
14              Image("grinning-face-with-star-eyes")
15                  .resizable()
16                  .frame(width: 60, height: 60)
17              Text("Face With Starry Eyes")
18          }
19      }
20  }
21
22  struct ContentView_Previews: PreviewProvider {
23      static var previews: some View {
```

Views displayed in each row

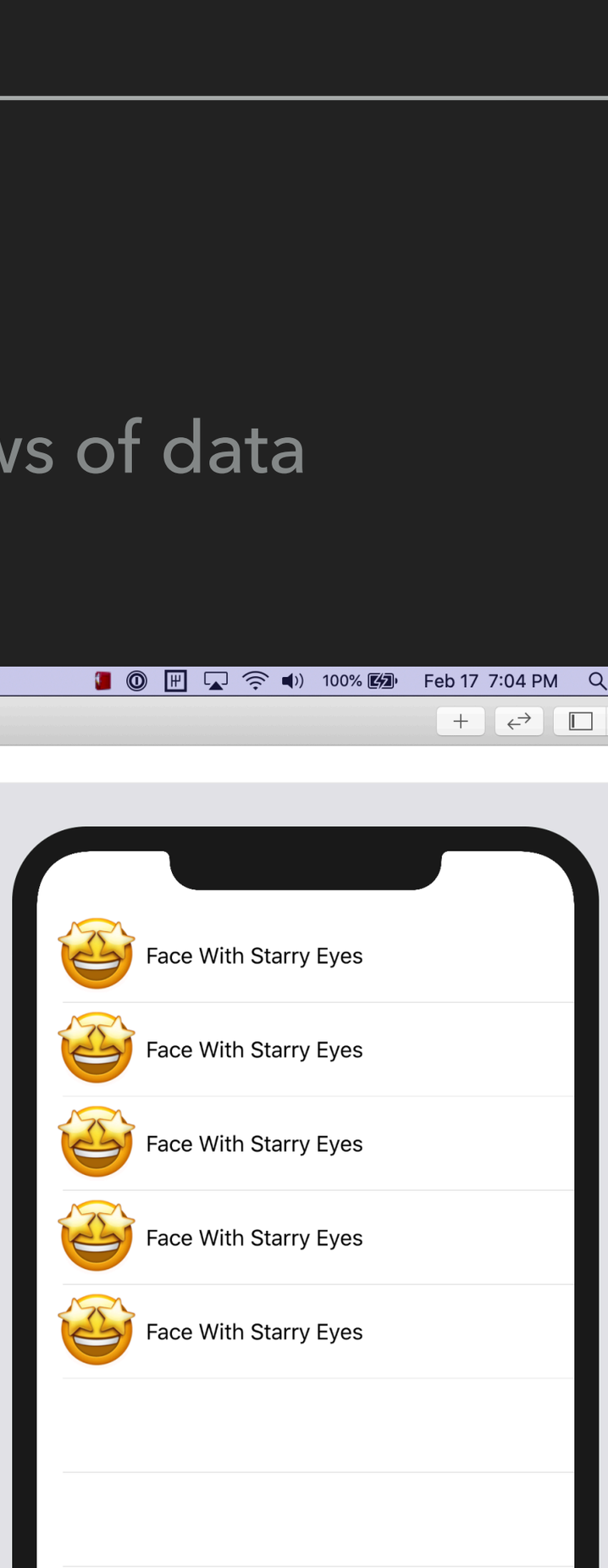


LISTS

- `List` allows you to present multiple rows of data


```
8
9   import SwiftUI
10
11  struct ContentView: View {
12      var body: some View {
13          List(0..<5) { item in
14              Image("grinning-face-with-star-eyes")
15                  .resizable()
16                  .frame(width: 60, height: 60)
17              Text("Face With Starry Eyes")
18          }
19      }
20  }
21
22  struct ContentView_Previews: PreviewProvider {
23      static var previews: some View {
```

Data for each row



LISTS

- SwiftUI needs a way to distinguish the model object for each row
    - Option 1: Provide a key path to a property that uniquely identifies each model object
    - Option 2: Have the model conform to the `Identifiable` protocol

LISTS

- Here is our `Emoji` model

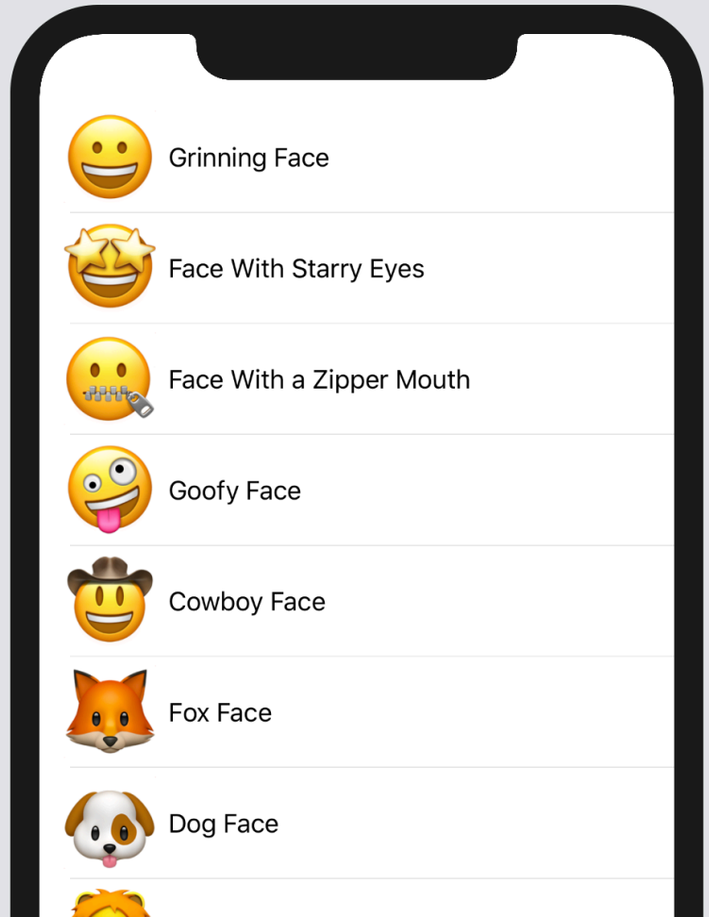
```
struct Emoji: Codable {
    let name: String
    let year: String
    let imageName: String
}
```

LISTS

- We can pass `name` as the identifier


```
8
9   import SwiftUI
10
11  struct ContentView: View {
12
13      let emojis: [Emoji] = emojiData()
14
15      var body: some View {
16          List(emojis, id: \.name) { emoji in
17              Image(emoji.imageName)
18                  .resizable()
19                  .frame(width: 60, height: 60)
20              Text(emoji.name)
21          }
22      }
23  }
24
```

Identifier



LISTS

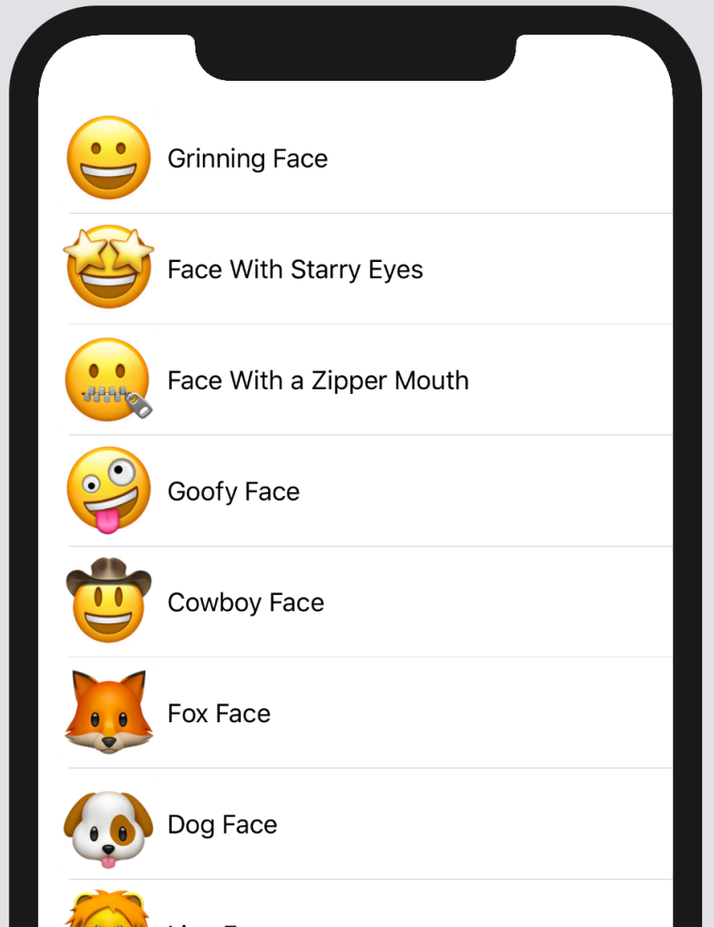
- Alternatively, we can update `Emoji` to conform to the `Identifiable` protocol

```
struct Emoji: Codable, Identifiable {
    let name: String
    let year: String
    let imageName: String
    let id: UUID = UUID()
}
```

LISTS

- Now we can pass in an array of emojis without needing to provide the `id` key path as well


```
8
9   import SwiftUI
10
11  struct ContentView: View {
12
13      let emojis: [Emoji] = emojiData()
14
15      var body: some View {
16          List(emojis) { emoji in
17              Image(emoji.imageName)
18                  .resizable()
19                  .frame(width: 60, height:
                        60)
20              Text(emoji.name)
21          }
22      }
23  }
```



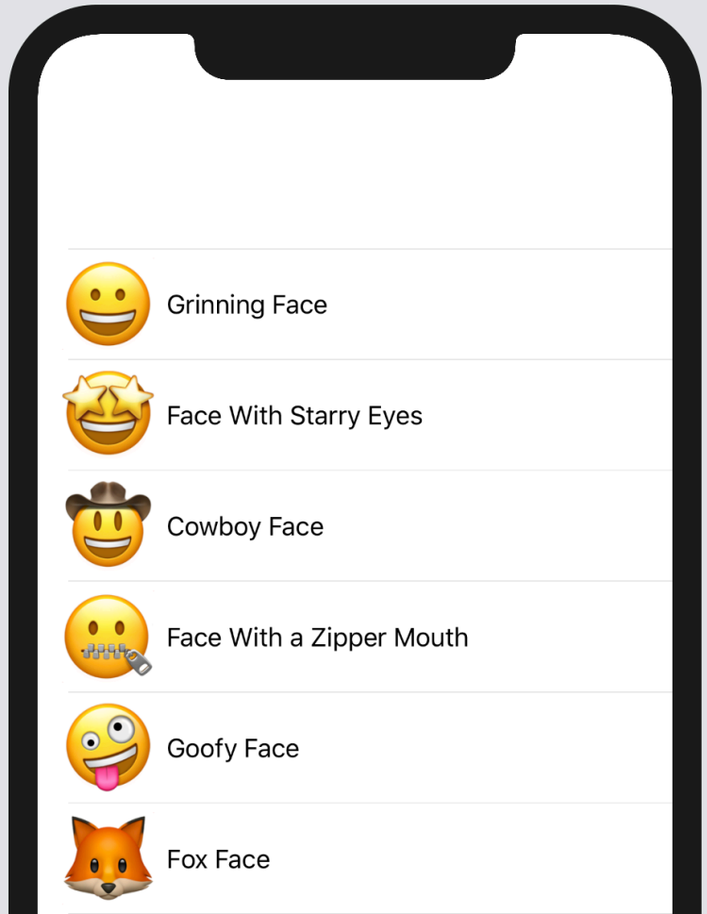
NAVIGATION VIEWS

- Suppose we want users to be able to tap on a row and navigate to a detail page
- This can be achieved using `NavigationView`

NAVIGATION VIEWS

- Embed the list in a `NavigationView`

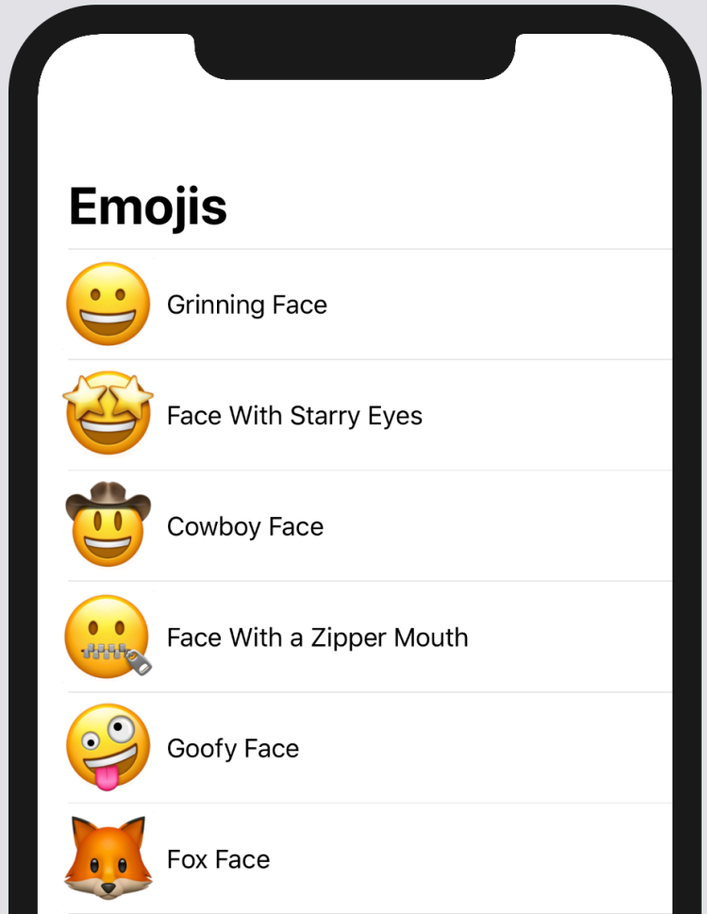

```
8
9   import SwiftUI
10
11  struct ContentView: View {
12
13      let emojis: [Emoji] = emojiData()
14
15      var body: some View {
16          NavigationView {
17              List(emojis) { emoji in
18                  EmojiRow(emoji: emoji)
19              }
20          }
21      }
22  }
23
24  struct ContentView_Previews: PreviewProvider {
```



NAVIGATION VIEWS

- Add a title using the `navigationBarTitle` modifier

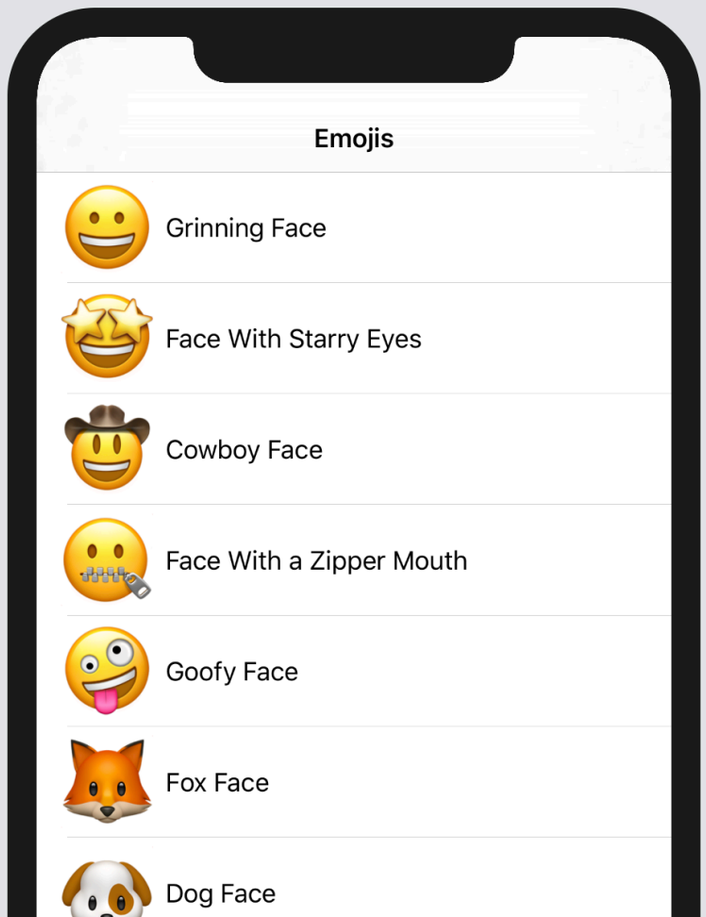

```
8
9   import SwiftUI
10
11  struct ContentView: View {
12
13      let emojis: [Emoji] = emojiData()
14
15      var body: some View {
16          NavigationView {
17              List(emojis) { emoji in
18                  EmojiRow(emoji: emoji)
19              }
20              .navigationBarTitle("Emojis")
21          }
22      }
23  }
24
```



NAVIGATION VIEWS

- Set the `displayMode` to inline if you don't want large titles


```
8
9   import SwiftUI
10
11  struct ContentView: View {
12
13      let emojis: [Emoji] = emojiData()
14
15      var body: some View {
16          NavigationView {
17              List(emojis) { emoji in
18                  EmojiRow(emoji: emoji)
19              }
20              .navigationBarTitle("Emojis", displayMode: .inline)
21          }
22      }
23  }
24
```



NAVIGATION VIEWS

- Add a `NavigationLink` so that tapping on a row triggers navigation to a new page


```
8
9   import SwiftUI
10
11  struct ContentView: View {
12
13      let emojis: [Emoji] = emojiData()
14
15      var body: some View {
16          NavigationView {
17              List(emojis) { emoji in
18                  NavigationLink(destination: Text("Hello")) {
19                      EmojiRow(emoji: emoji)
20                  }
21              }
22              .navigationBarTitle("Emojis", displayMode: .inline)
23          }
24      }
```

HANDLING DATA IN

SWIFTUI

DATA

- Data is the information that drives your UI
  - Array of model objects –> `List`
  - Boolean –> `Toggle`

DATA AS A DEPENDENCY

- When a view needs to access a piece of data, you create a dependency between the view and the data
- Each time the data changes, you need to update the view

DATA AS A DEPENDENCY

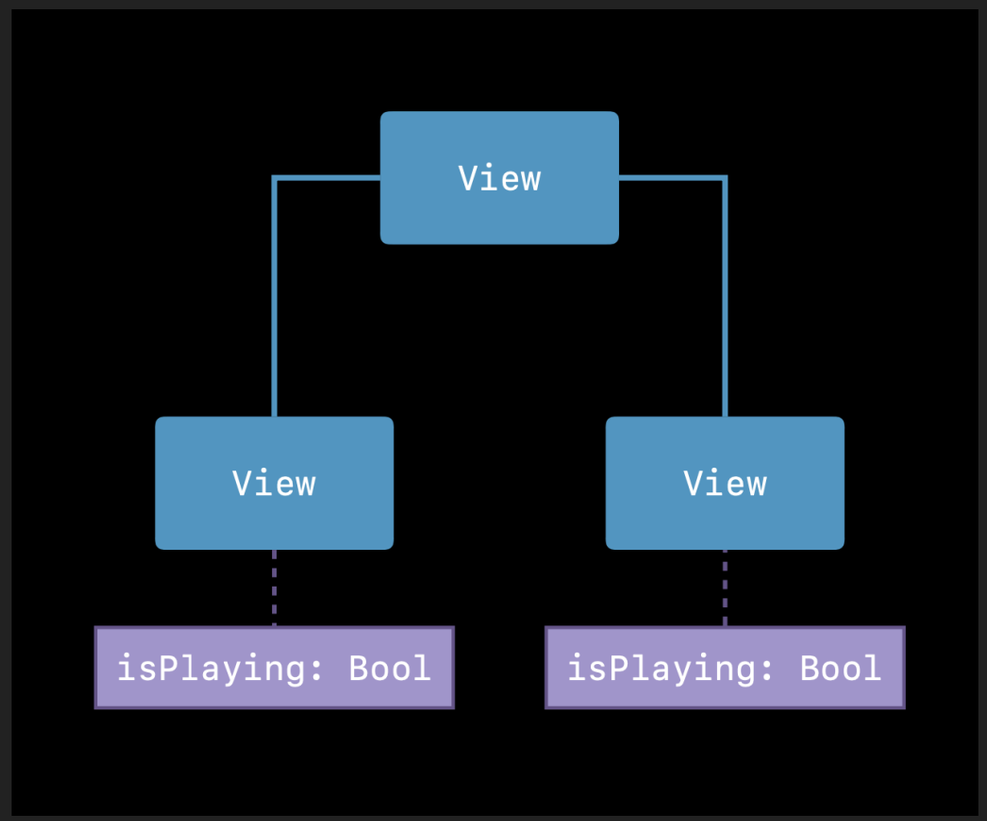
- With UIKit, keeping view and data in sync requires manual effort
    - If the data changes, you need to update the view
    - If the view changes, you need to update the data

DATA AS A DEPENDENCY

- With SwiftUI, data dependency is declarative
- You describe the dependency to the framework and the framework handles keeping data and view in sync

SOURCE OF TRUTH

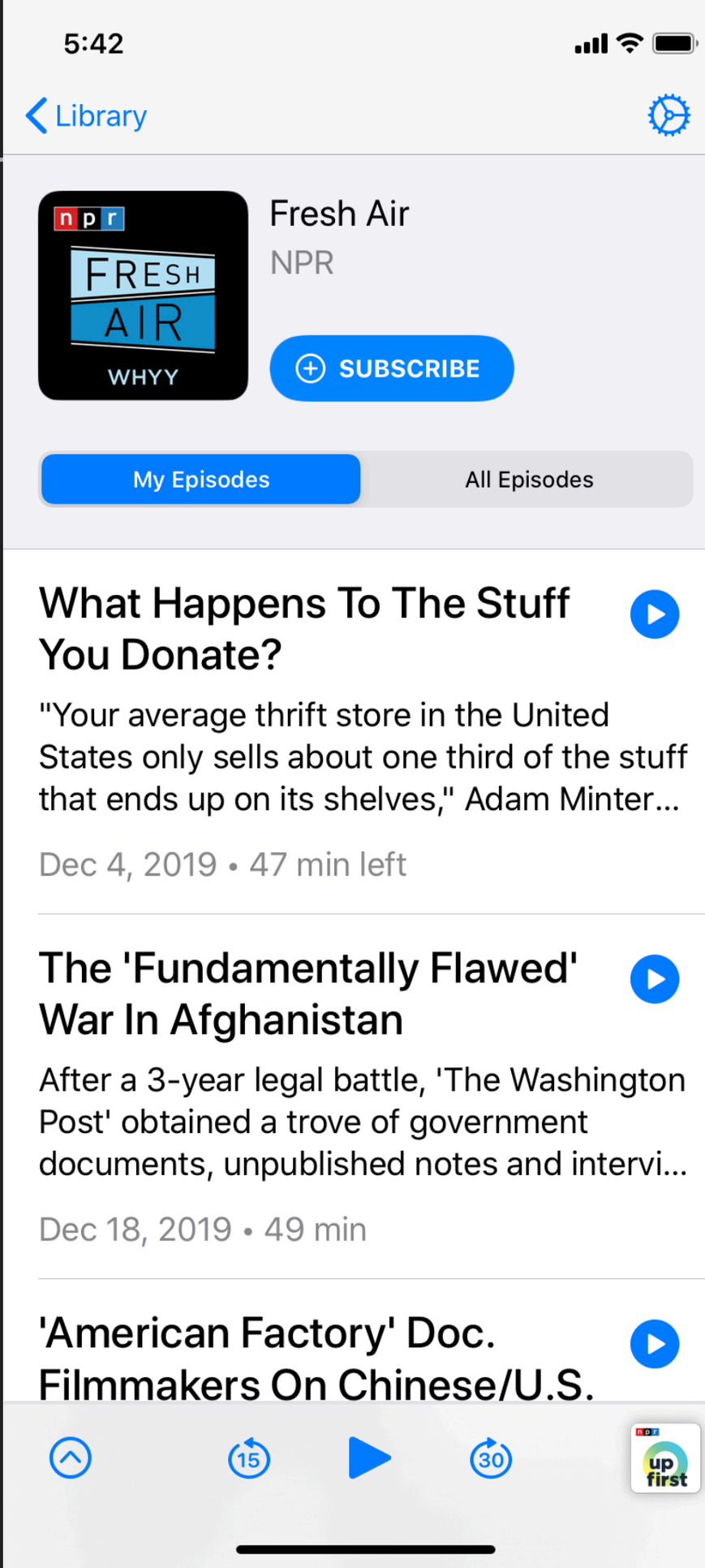
- Every piece of data should have a single source of truth
- Keeping duplicate copies of data is a common source of bugs



SOURCE OF TRUTH

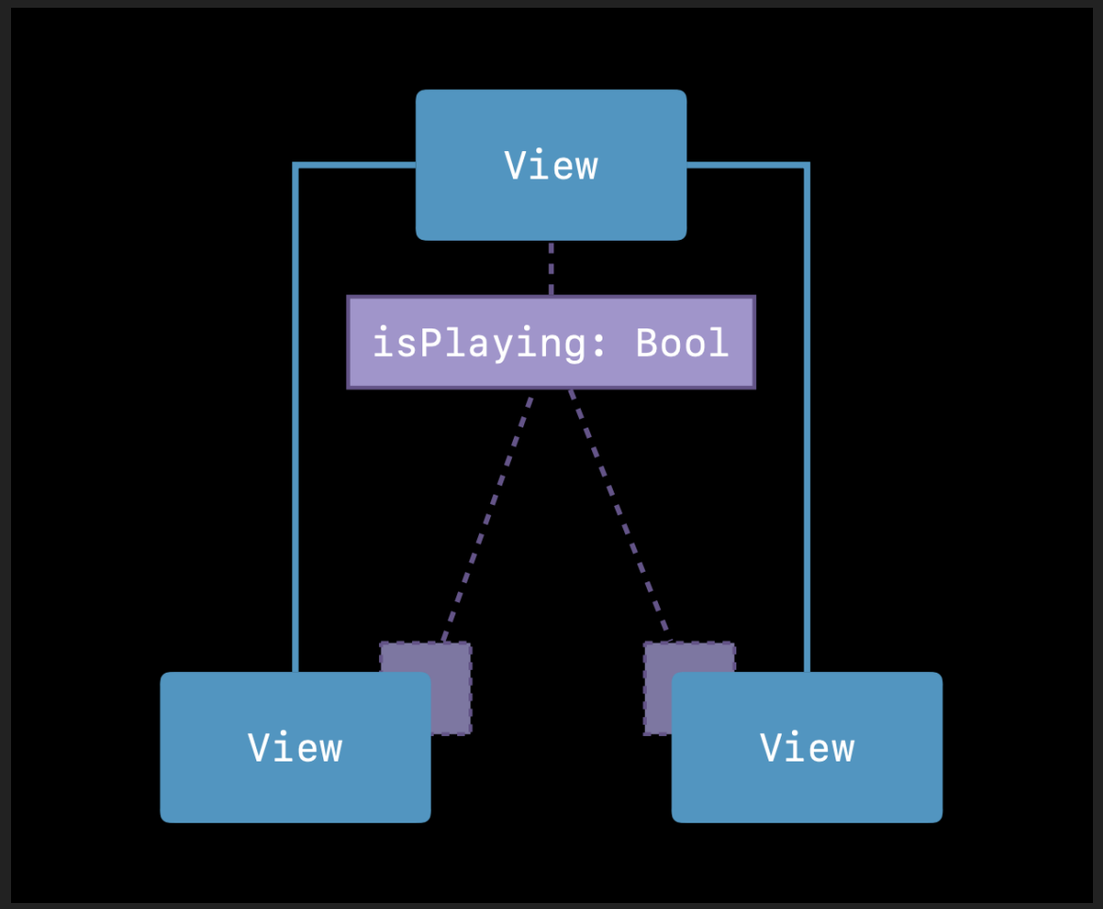
- Example:
  - Playback control at the bottom of the screen
  - Play button in each row

Castaway



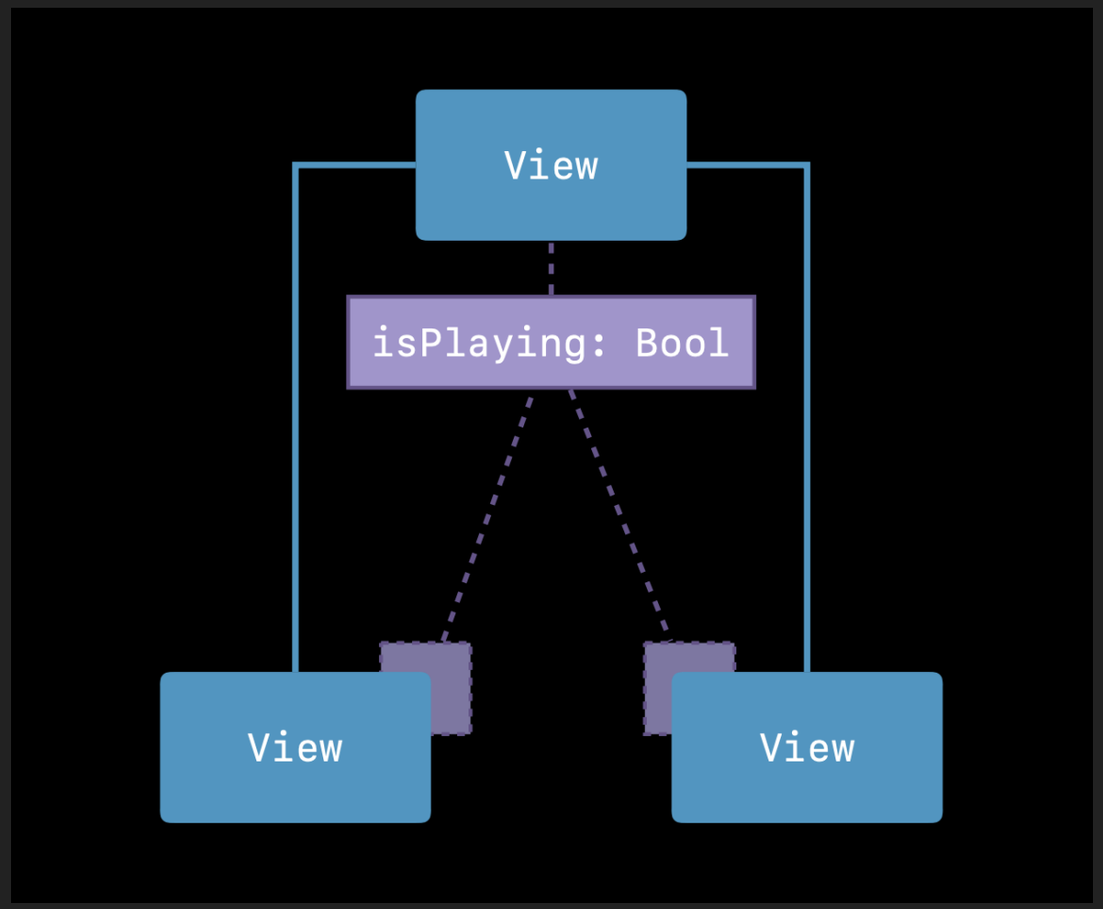
SOURCE OF TRUTH

- SwiftUI's solution: maintain a single source of truth in a common ancestor and let two child views have a reference to it



SOURCE OF TRUTH

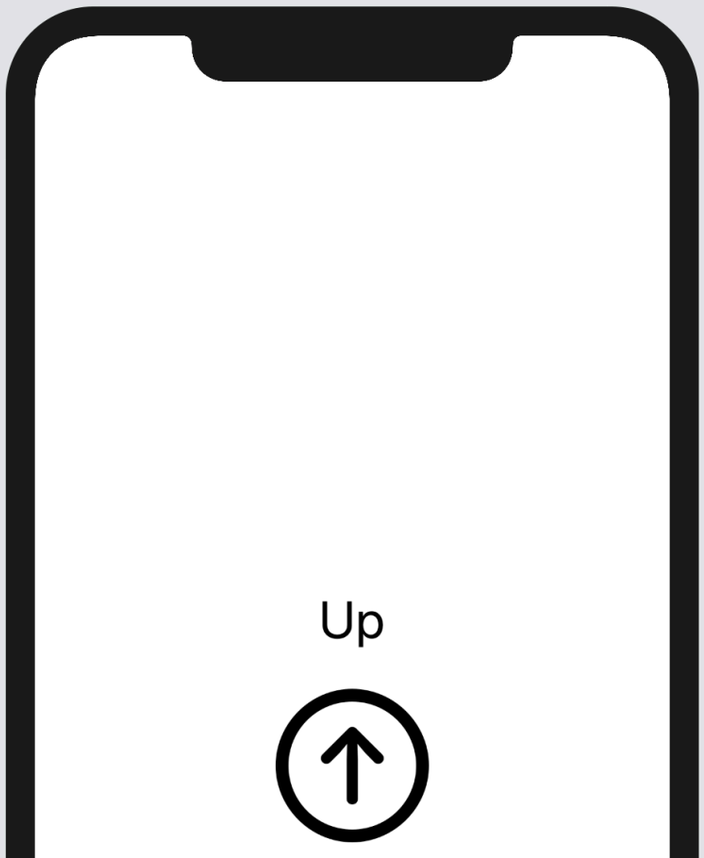
- SwiftUI's solution: maintain a single source of truth in a common ancestor and let two child views have a reference to it



HANDLING DATA

- If the view needs read-only access to a piece of data, use a simple Swift property

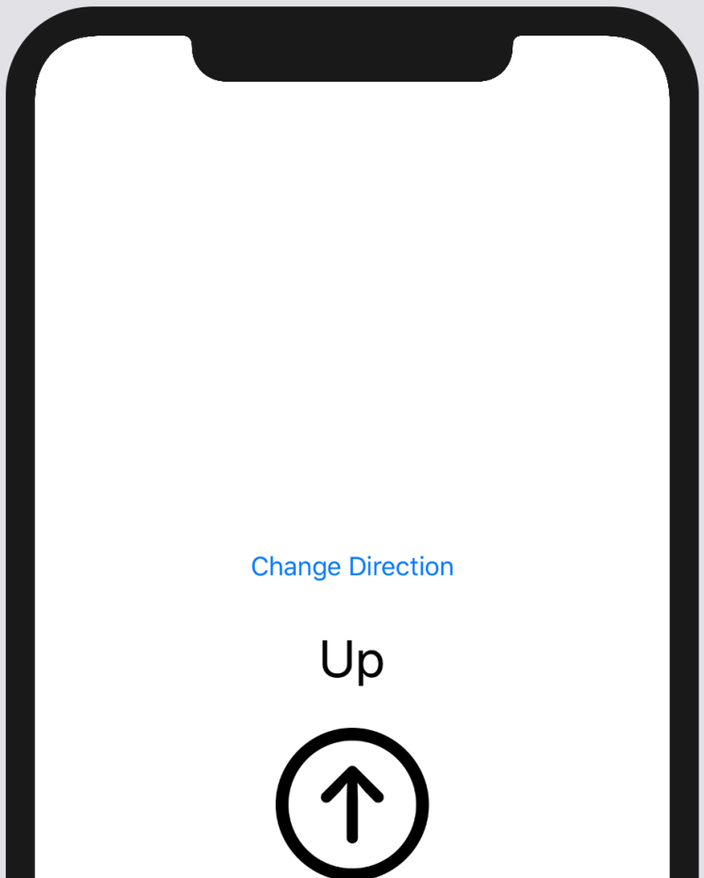

```
8
9   import SwiftUI
10
11  struct ArrowView: View {
12
13      let pointsUp: Bool
14
15      var body: some View {
16          VStack {
17              Text(self.pointsUp ? "Up" : "Down")
18                  .font(.largeTitle)
19              Image(systemName: self.pointsUp ?
                    "arrow.up.circle" : "arrow.down.circle")
20                  .resizable()
21                  .frame(width: 100, height: 100)
22                  .padding(.bottom, 20)
```



HANDLING DATA

- Let's add a button to this view and let it determine the direction the arrow is pointing


```
8
9   import SwiftUI
10
11  struct ArrowView: View {
12
13      let pointsUp: Bool
14
15      var body: some View {
16          VStack {
17              Button(action: {
18                  // TODO
19              }, label: {
20                  Text("Change Direction")
21              })
22              Text(self.pointsUp ? "Up" : "Down")
23                  .font(.largeTitle)
```



HANDLING DATA

- Simply changing the boolean in the button's action won't work

SwiftUIDemo 〉 iPhone 11 Pro Max | SwiftUIDemo | Preview SwiftUIDemo: **Succeeded** | Today at 6:11 AM

SwiftUIDemo 〉 SwiftUIDemo 〉 ArrowView.swift 〉 No Selection

```
8
9   import SwiftUI
10
11  struct ArrowView: View {
12
13      var pointsUp: Bool = true
14
15      var body: some View {
16          VStack {
17              Button(action: {
18                  self.pointsUp.toggle()
19              }, label:
20                  Text("
21              })
22              Text(self.pointsUp ? "Up" : "Down")
23                  .font(.largeTitle)
```

Cannot use mutating member on immutable value: 'self' is immutable

Automatic preview updating paused Resume

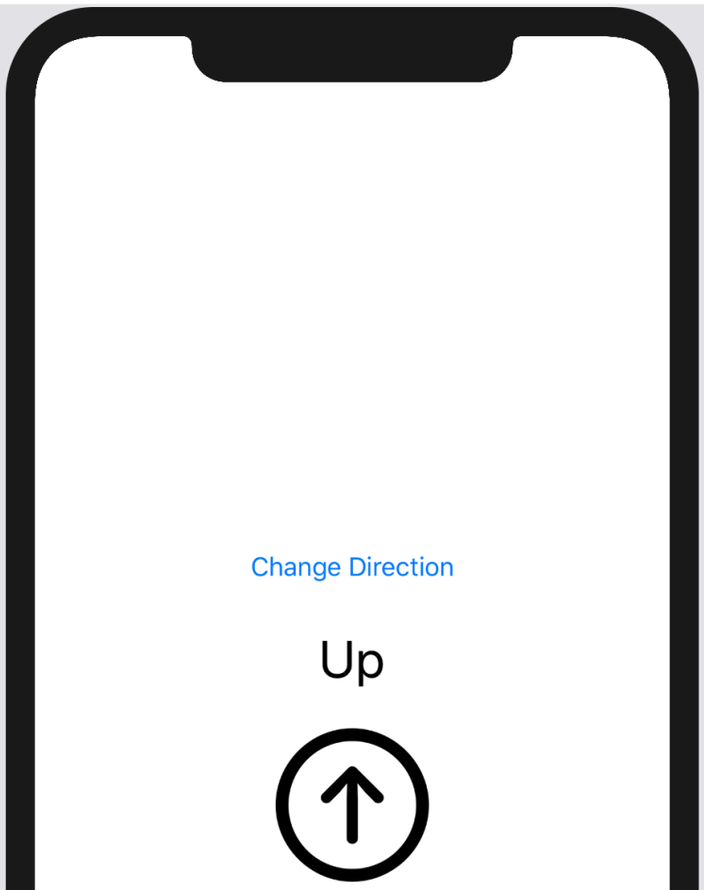
Change Direction

Up

HANDLING DATA

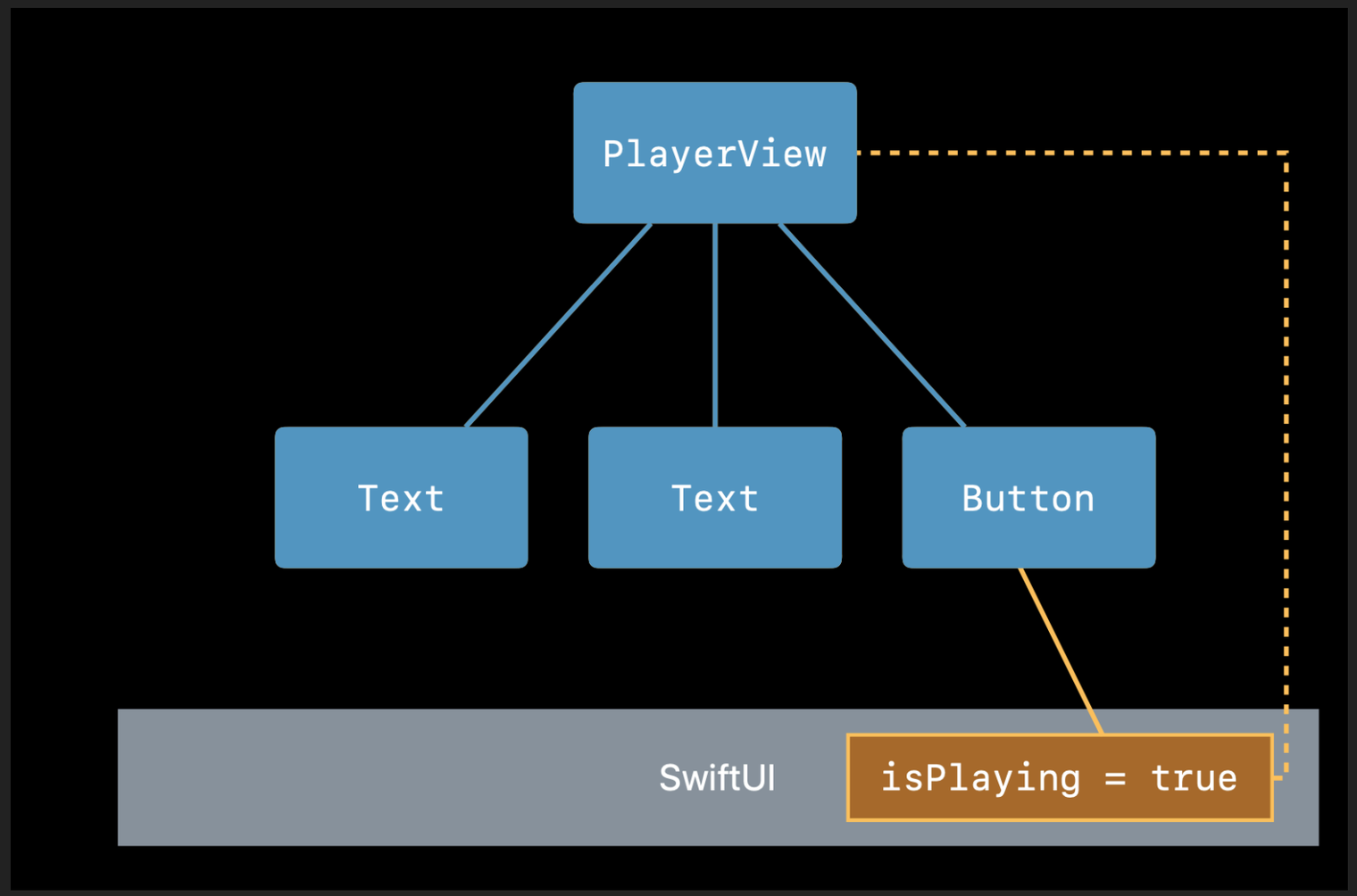
- We need to mark it with the `@State` attribute


```
8
9   import SwiftUI
10
11  struct ArrowView: View {
12
13      @State var pointsUp: Bool = true
14
15      var body: some View {
16          VStack {
17              Button(action: {
18                  self.pointsUp.toggle()
19              }, label: {
20                  Text("Change Direction")
21              })
22              Text(self.pointsUp ? "Up" : "Down")
23                  .font(.largeTitle)
```



HANDLING DATA

- When a property marked with @State is updated, the change flows down through the view hierarchy



VIEWS ARE A FUNCTION OF STATE, NOT OF A SEQUENCE OF EVENTS.

HANDLING DATA

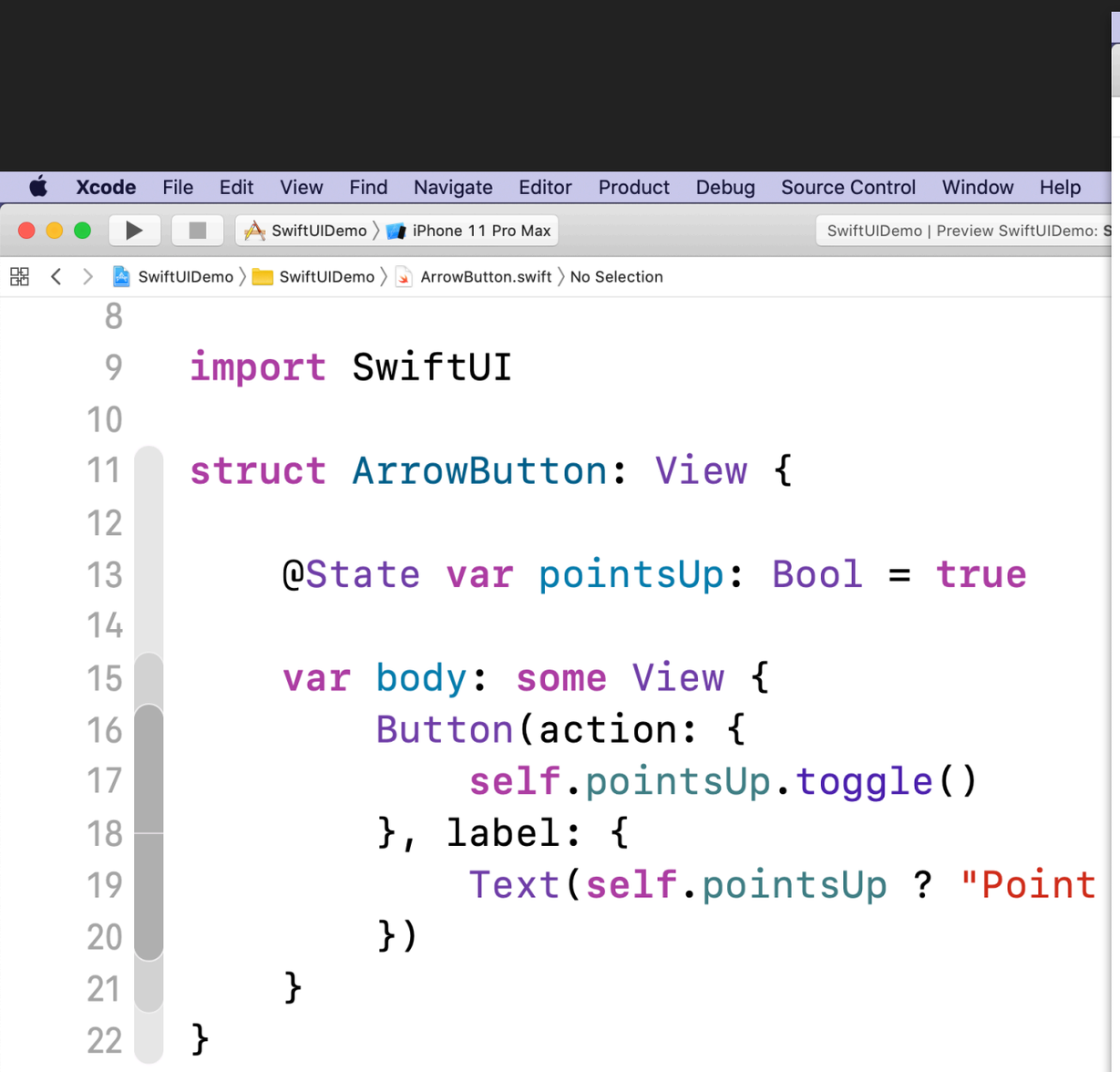
- Suppose we want to extract button code into its own view

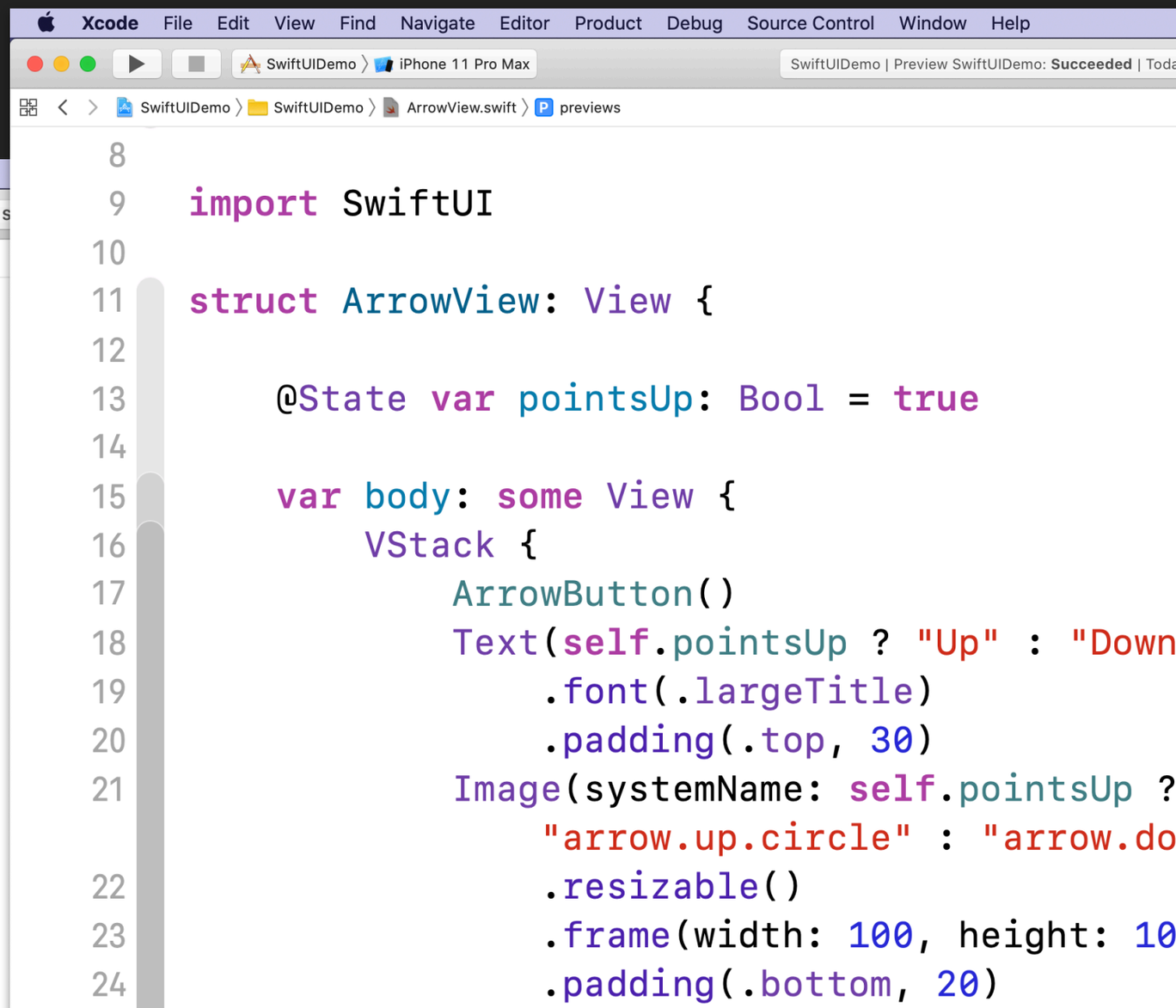

```
8
9   import SwiftUI
10
11  struct ArrowButton: View {
12
13      @State var pointsUp: Bool = true
14
15      var body: some View {
16          Button(action: {
17              self.pointsUp.toggle()
18          }, label: {
19              Text(self.pointsUp ? "Point Down" : "Point Up")
20          })
21      }
22  }
```

Point Down

HANDLING DATA

- But wait! Now we have two sources of truth for `pointsUp`

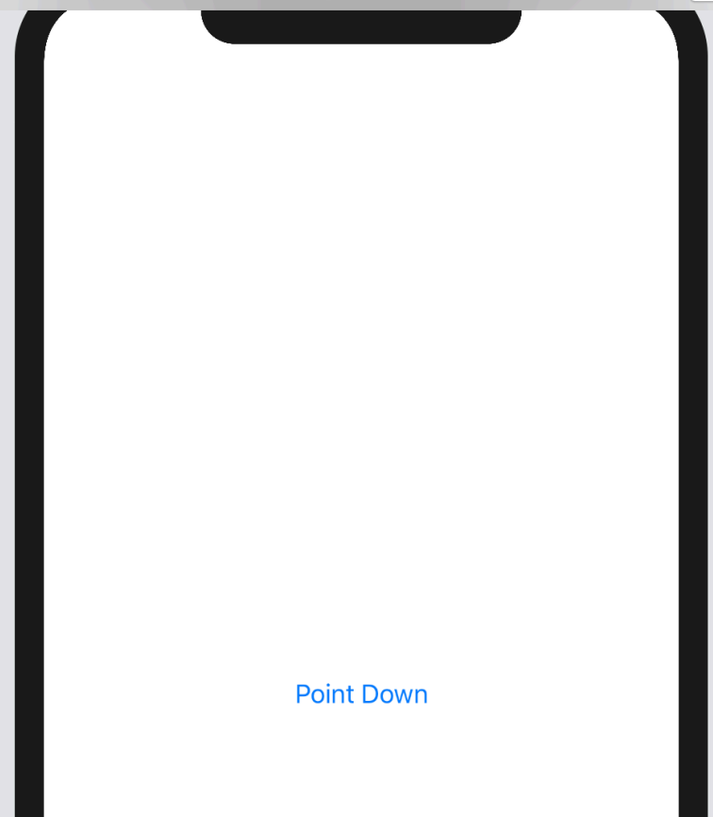




HANDLING DATA

- Instead we can use `@Binding` to allow a view to read and write a value that it doesn't own


```
8
9   import SwiftUI
10
11  struct ArrowButton: View {
12
13      @Binding var pointsUp: Bool
14
15      var body: some View {
16          Button(action: {
17              self.pointsUp.toggle()
18          }, label: {
19              Text(self.pointsUp ? "Point Down" : "Point Up")
20          })
21      }
22  }
```

HANDLING DATA

- `@State` is useful when:
  - The view determines the value of the property
  - The data is a simple type (Boolean, Int, etc.)
- What if the data is more complex and determined by something external to the view?
  - Example: Network request

HANDLING DATA

- To represent more complex data, create a class that conforms to the `ObservableObject` protocol

```
class GitHubIssues: ObservableObject {

}
```

HANDLING DATA

- Use `@Published` to indicate properties that should cause views to re-render when they change

```
class GitHubIssues: ObservableObject {

    @Published var openIssues: [GitHubIssue] = []
    @Published var closedIssues: [GitHubIssue] = []

}
```

HANDLING DATA

- Whenever `@Published` properties are updated, SwiftUI will automatically re-draw your views with the new data

```
class GitHubIssues: ObservableObject {

    @Published var openIssues: [GitHubIssue] = []
    @Published var closedIssues: [GitHubIssue] = []

    init() {
        // Fetch data using URLSession
    }
}
```

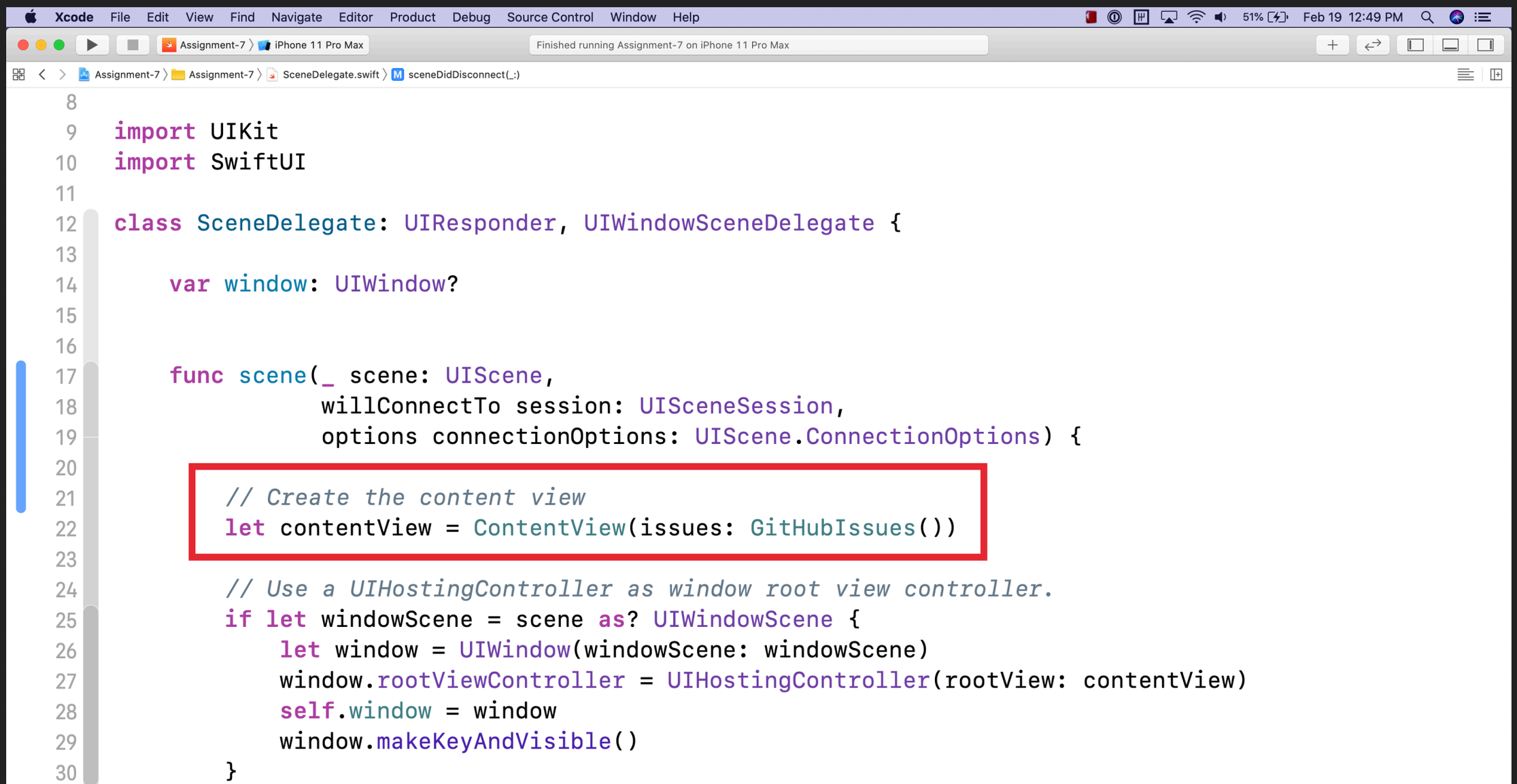
HANDLING DATA

- To reference an ObservableObject in a view, use `@ObservedObject`

```
import SwiftUI

struct ContentView: View {

    @ObservedObject var issues: GitHubIssues

    var body: some View {
        // Define the view's body
    }
}
```

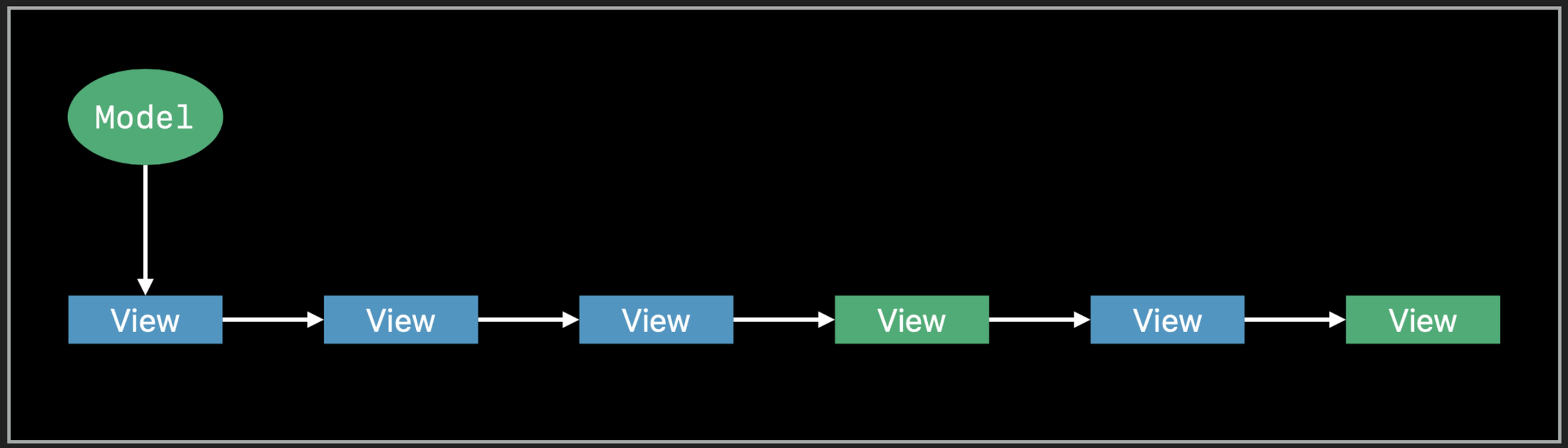
HANDLING DATA

- Pass the `ObservableObject` into the view's initializer



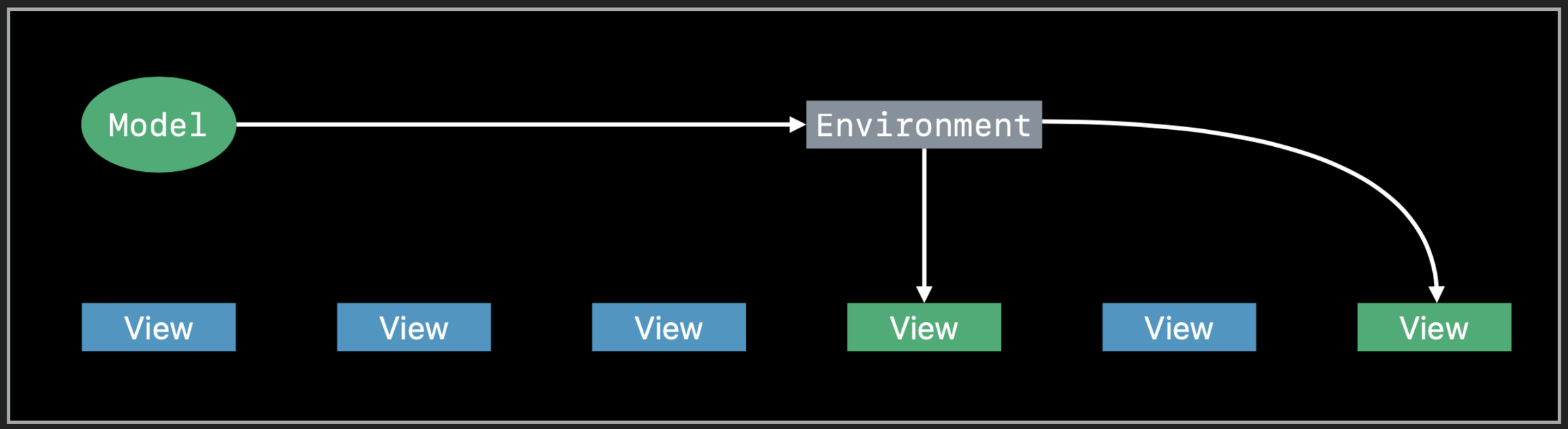
HANDLING DATA

- Continue passing the `ObservableObject` into the initializer for any view that requires it



HANDLING DATA

- Alternatively, use `@EnvironmentObject` so that any view can access the data indirectly



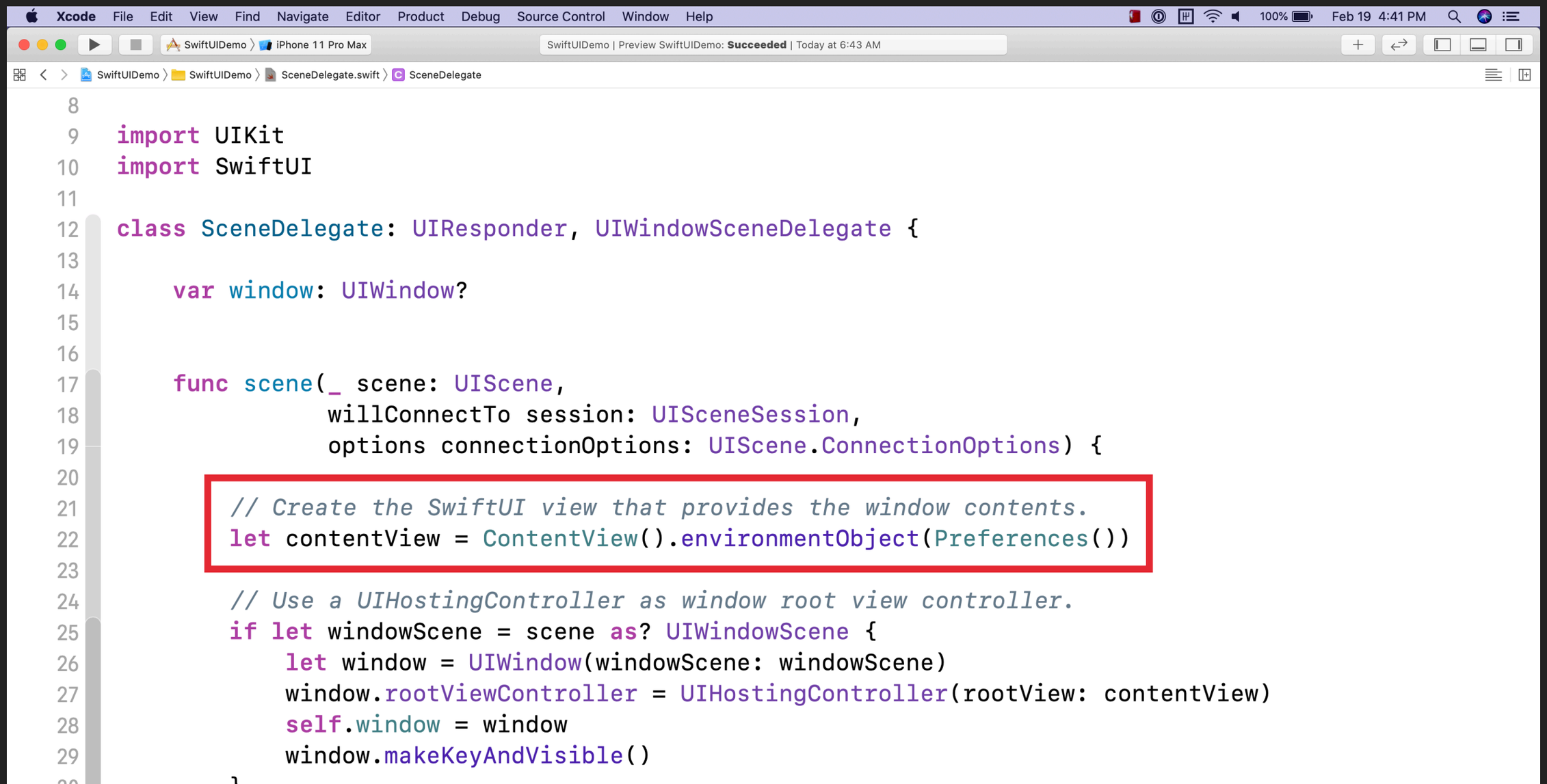
HANDLING DATA

- Your `ObservableObject` stays the same

```
class GitHubIssues: ObservableObject {

    @Published var openIssues: [GitHubIssue] = []
    @Published var closedIssues: [GitHubIssue] = []

    init() {
        // Fetch data using URLSession
    }
}
```

HANDLING DATA

- Use the `.environmentObject` method to set the data



SUMMARY

- `@State` - simple properties owned by a single view
- `@ObservedObject` - complex properties shared across several views
- `@EnvironmentObject` - complex properties accessible to all views

SUMMARY

- `@Binding` - gives a view read/write access to a `@State` property that it doesn't own

SUMMARY

- `ObservableObject` - protocol that data models must conform to in order to be used with @ObservedObject or @EnvironmentObject
- `@Published` - property on an ObservableObject
  - Views using an @Published property will re-render each time the property changes

ROUGH EDGES

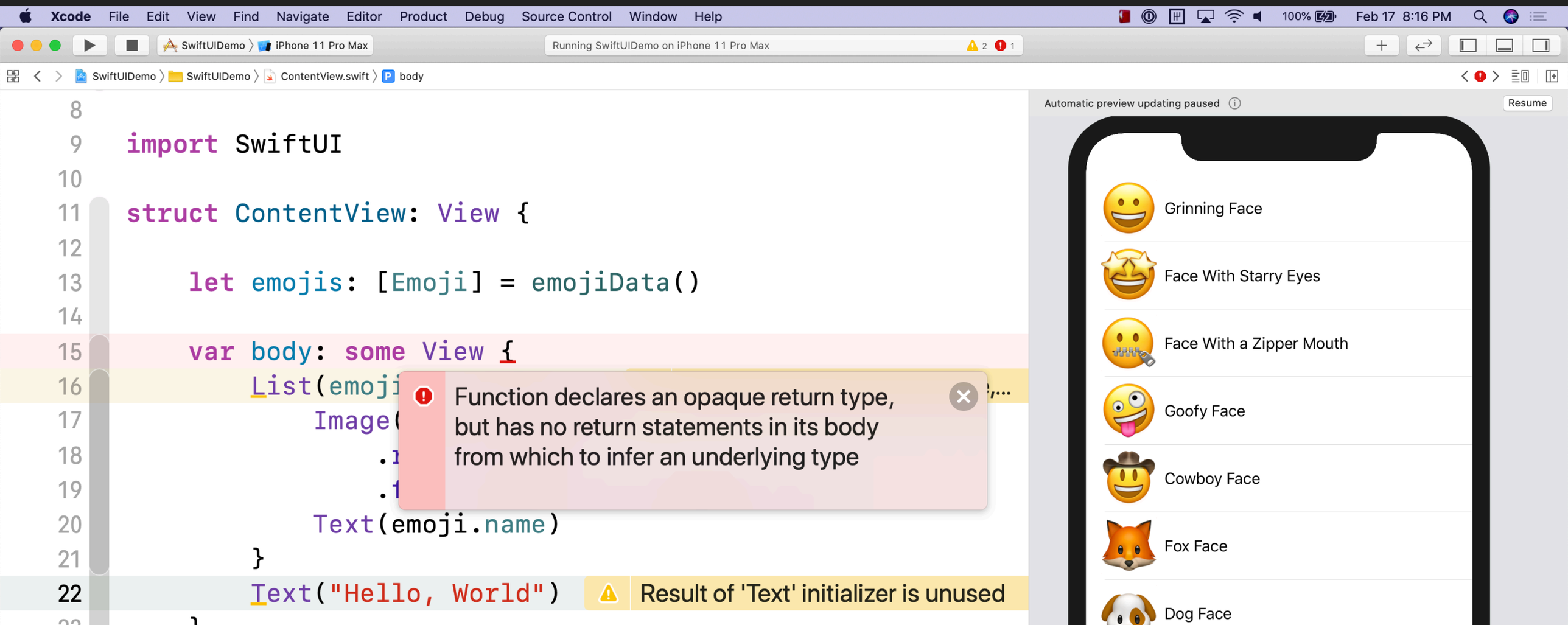
SWIFTUI

ROUGH EDGES

- As you're working with SwiftUI, keep in mind that it's a brand new API
- This means there are a few rough edges

ROUGH EDGE #1

- Error messages are sometimes unhelpful
- They may even appear on the wrong line



ROUGH EDGE #2

- The official documentation is spotty at best

No Overview Available.

A survey of Apple developer documentation.

Framework
SwiftUI

4829 / 11901 (40.6%) Documented

7072 Undocumented Symbols

- !=(_:_:) AccessibilityActionKind
- !=(_:_:) AccessibilityChildBehavior
- !=(_:_:) Alignment
- !=(_:_:) AnimatablePair

ROUGH EDGE #3

- Live Previews require a lot of CPU power, so you might see your computer start to lag

ASSIGNMENT #7

GITHUB ISSUES

GITHUB ISSUES

- Display open and closed GitHub Issues… using SwiftUI!

GITHUB ISSUES

- Due Wednesday, February 26 at 5:29pm
- Post any questions to Slack